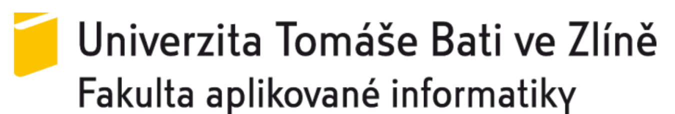
Detekce mobilního malwaru na základně signatur netriviálních příznaků

Ladislav Dorotík

Bakalářská práce
2020



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Ladislav Dorotík**
Osobní číslo:	**A17012**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Detekce mobilního malwaru na základě signatur netriviálních příznaků**
Téma práce anglicky:	**Mobile Malware Detection Based on Non-trivial Symptom Signatures**

Zásady pro vypracování

1. Rozšiřte základní jednoduché detekce mobilního malwaru, které probíhají na základě kontroly package name a hash hodnoty APK balíčku.
2. Pro vypracování použijte databázi vzorků mobilního malwaru, kterou poskytne společnost Monet+.
3. Navrhněte další příznaky, které lze použít pro klasifikaci aplikace jako malwaru. Součástí práce bude i návrh, jak tyto příznaky využít pro klasifikaci mobilního malwaru.
4. Vytvořte implementaci navrženého klasifikátoru jako modulu do SDK pro detekci mobilního malwaru na platformě Android.

Rozsah bakalářské práce:
Rozsah příloh:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. AU, Man Ho. Mobile security and privacy: advances, challenges and future research directions. Cambridge, MA: Elsevier, 2016. ISBN 978-012-8046-296.
2. DUNHAM, Ken, Shane HARTMAN, Jose Andre MORALES, Manu QUINTANS a Tim STRAZZERE. Android malware and analysis. Boca Raton: Auerbach Publications, [2014]. ISBN 14-822-5219-8.
3. DUBEY, Abhishek a Anmol MISRA. Android security: attacks and defenses. Boca Raton: CRC Press, c2013. ISBN 978-1-4398-9646-4.
4. JIANG, Xuxian a Yajin ZHOU. Android malware. New York: Springer, [2013]. SpringerBriefs in computer science. ISBN 978-1-4614-7393-0.
5. ELENKOV, Nikolay. Android security internals: an in-depth guide to Android's security architecture. San Francisco: No Starch Press, [2015]. ISBN 15-932-7581-1.
6. RAGGO, Michael T. Mobile data loss: threats and countermeasures. Waltham, MA: Syngress, [2016]. ISBN 01-280-2864-5.
7. DUNHAM, Ken. Mobile malware attacks and defense. Burlington, MA: Elsevier, c2009. ISBN 15-974-9298-1.

Vedoucí bakalářské práce: **Ing. Milan Oulehla**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **28. listopadu 2019**
Termín odevzdání bakalářské práce: **15. května 2020**

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

Ladislav Dorotík:

Detekce mobilního malwaru na základě signatur netriviálních příznaků:

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl jsem seznámen s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 10. 8. 2020

Ladislav Dorotík, v. r.
podpis diplomanta

ABSTRAKT

Práce se zabývá detekcí mobilního malware na platformě Android OS. Cílem práce bylo navrhnutí příznaků použitelných jako signatur a vytvoření detekční aplikace pro ověření správné funkcionality. Detekční aplikace je realizována vícestupňovou ochranou, ve které se porovnává shoda hašovací funkce, oprávnění, uses-features a komponent jednotlivých aplikací. Příznaky byly vytvořeny na základě provedené statické analýzy malwaru. Statická analýza byla provedena na APK souborech z databáze malwarů.

Detekční aplikace dosahuje stoprocentní úspěšnosti u známých vzorků a do jisté míry je schopna reagovat na obfuskační techniky a vzorky neznámé. Všechny nainstalované vzorky malware byly úspěšně detekovány a odinstalovány.

Klíčová slova: Android, APK, databáze, detekce, hašovací funkce, komponenta, malware, obfuskační techniky, oprávnění, příznaky, signatury

ABSTRACT

This work is concentrated on mobile malware detection which operates on the Android OS platform. The main goal of work was to find out symptoms, which can be used as signatures, and to create a detection application to test the correct functionality. Detection application is based on multi-level protection that compares the match of a hash function, permissions, uses-features, and components of each app. The signatures were designed based on static analysis. Static analysis was conducted on APK files from the malware database.

Detection application reaches one hundred percent success with known samples and is slightly able to react to obfuscation techniques and unknown samples. All installed samples have been detected and uninstalled successfully.

Keywords: Android, APK, component, database, detection, hash function, malware, obfuscation techniques, permissions, signatures, symptoms

Chtěl bych poděkovat Mgr. Anežce Pejlové, Mgr. Ondřeji Gabrhelíkovi a Bc. Ondřeji Přikrylovi ze společnosti AHEAD iTec (dceřiná společnost Monet+) za vypsání tématu, poskytnutí databáze malwaru, také za všechny uskutečněné schůzky a videokonference, které pro mě byly vždy přínosem.

Speciální poděkování patří Ing. Milanu Oulehlovi, Ph.D., který mi věnoval velké množství svého času, svých zkušeností a znalostí. Hlavně mě ale inspiroval k tématu kybernetické bezpečnosti a ukázal mi svůj pohled na informační technologie, za což jsem nezměřitelně vděčný.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD **9**
I TEORETICKÁ ČÁST **11**
1 PLATFORMA ANDROID **12**
1.1 Architektura operačního systému Android 12
1.1.1 Linuxové jádro 13
1.1.2 Abstraktní vrstva HAL 13
1.1.3 Běhové prostředí ART (Dalvik Virtual Machine) 13
1.1.4 Nativní knihovny C/C++ 14
1.1.5 Aplikační framework Java API 14
1.1.6 Systémové a uživatelské aplikace 15
1.2 Aplikace běžící v Android OS 15
1.2.1 Struktura aplikace 15
1.2.2 Hlavní komponenty aplikace 17
1.2.3 Podepisování aplikace 22
2 MALWARE A STATICKÁ ANALÝZA **25**
2.1 Malware 25
2.1.1 Distribuce malwaru 25
2.1.2 Využití malware 27
2.2 Statická analýza 29
2.2.1 Obfuskační techniky 29
3 DETEKCE MALWARE **31**
3.1 Obecné rozdělení detekce malware 31
3.2 Rozdělní dle zkoumaných dat 32
3.2.1 Permission-based 32
3.2.2 AndroidManifest-based 32
3.2.3 API-based 32
3.2.4 Certification-based 33
II PRAKTICKÁ ČÁST **34**
4 STATICKÁ ANALÝZA VZORKŮ MALWARE **35**
4.1 Datová sada 35
4.2 Statická analýza APK 36
4.2.1 Nástroje 36
4.2.2 Ukázka statické analýzy APK – testovací případ SauronLocker 41
4.3 Navrhnutí příznaků použitelných pro klasifikaci malwaru 45
5 DETEKCE MALWARU **49**
5.2 Implementace 50
5.2.2 Modul A – Tvorba signatur z aplikace na zařízení 52

5.2.3 Modul B – Načtení signatur z otisků aplikací v databázi ... 57
5.2.4 Modul C – Detekce a odstranění malware ... 58
5.2.5 Další části detekční aplikace ... 61
5.3 TESTOVÁNÍ ... 63
5.4 VERZOVÁNÍ ... 64
ZÁVĚR ... 65
SEZNAM POUŽITÉ LITERATURY ... 67
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 74
SEZNAM OBRÁZKŮ ... 75
SEZNAM PŘÍLOH ... 77

ÚVOD

Technologie provází naši společnost na každém kroku a usnadňuje celou řadu každodenních činností, ale s každou novou technologií přichází i nová bezpečnostní rizika, a jinak tomu není ani ve výpočetní technice [1]. Uživatelé z celého světa se rychle přesunuli z klasických mobilních zařízení na chytrá zařízení [2].

Vzhledem k tomu, že se v posledních letech zvýšil počet chytrých telefonů na 3,5 bilionu [3], se stávají tato zařízení terčem útočníků čím dál častěji a jsou stále vyvíjeny nové škodlivé aplikace, tedy malware[1]. Jen v roce 2018 se množství nových variant malwarů zvětšilo o 54 %. [4] V dubnu roku 2019 bylo 75,3 % mobilních požadavků na webové stránky realizováno ze zařízení s operačním systémem Android [5]. Internetové požadavky jsou velice zajímavým ukazatelem, jelikož až 99,9 % mobilního malwaru pochází z obchodů třetích stran – tedy z internetu [4].

V praxi to znamená, že operační systém Android je nejčastěji vyskytující se systém, který i přesto, že zaznamenává pokles, má stále nejvyšší podíl na trhu a sice až 70,68 % [6]. Mobilní telefony a tablety navíc zaznamenávají až 56,66 % [7] podílu na trhu a o faktu, že jsou stále chytré telefony na vzestupu svědčí i to, že mobilní požadavky na webové stránky tvořily spolu s tablety 52,9 % z celkového provozu na internetu, tedy většina požadavků směřuje z mobilních telefonů a tabletů, zbylých 47,11 % tvoří počítače, notebooky, herní konzole a další zařízení [5].

Proč se jako tvůrce mobilního malwaru zaměřit na mobilní zařízení, potažmo Android OS je zřejmé – vzhledem k jeho otevřenosti spojené s množstvím zařízení, které jej využívají se stává ideálním cílem. Otázkou ovšem je, jak vytvářet mobilní malware a jaký důvod může člověk mít pro to, aby ho vytvořil. Problémem je, že tvůrce mobilního malwaru většinou nebývá průměrný programátor, ale spíše velmi kvalitní vývojář což znamená nejen, že tvoří škodlivé aplikace, ale snaží se je navíc často skrývat za použití různých zajímavých postupů, jak se detekci vyhnout. Pracovní pozice vývojáře patří mezi finančně náročné profese, takže proč by jej někdo platil, aby vytvořil škodlivou aplikaci, namísto velké produkční aplikace. Pro detekci malwaru je nutné pochopit nejen samotný operační systém a aplikace, ale i motivaci k jeho tvorbě.

[1] Obecný název pro všechny druhy škodlivých aplikací, vysvětleno v kapitole 2.

Výše uvedené skutečnosti naznačují aktuálnost tématu zabývající se mobilním malwarem zaměřeným na operačním systém Android. Práce se svými výsledky snaží přispět ke zlepšení bezpečnostní situace tím, že ukazuje možnosti zabezpečení a poukazuje na některé zranitelnosti aplikací v Android OS, zároveň práce upozorňuje na techniky, jakými se tvůrci mobilního malware pokouší své aplikace skrýt před detekčními systémy.

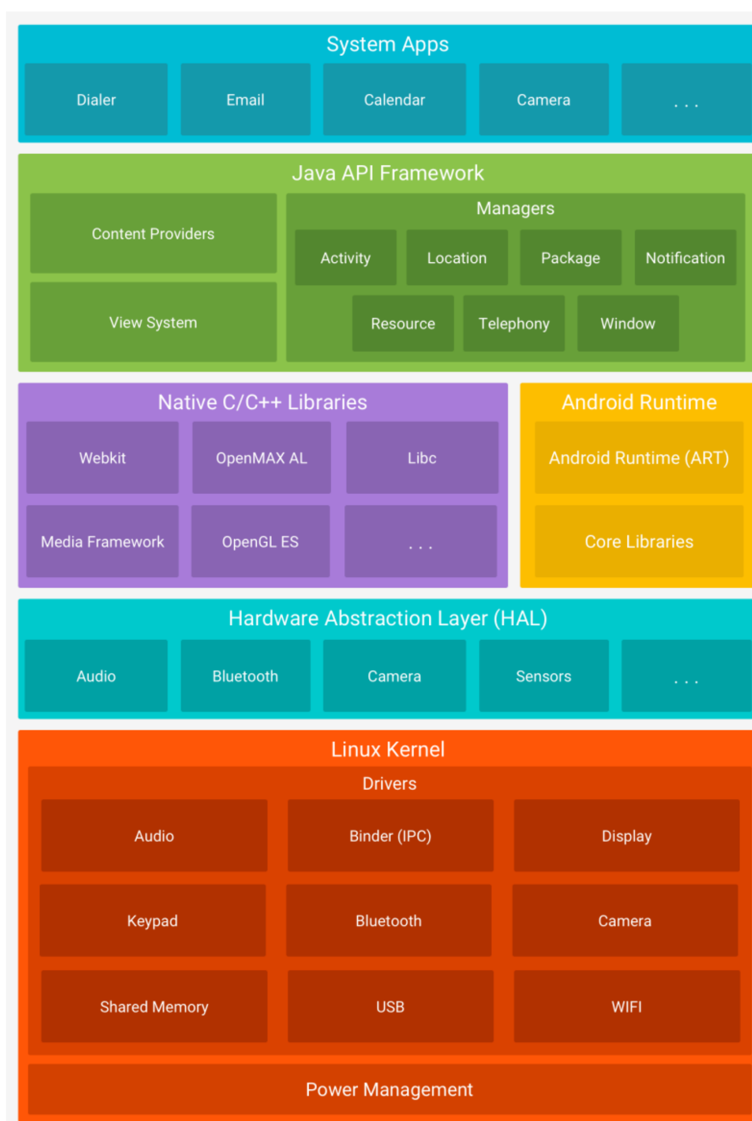
I. TEORETICKÁ ČÁST

1 PLATFORMA ANDROID

Operační systém Android je volně šiřitelná platforma založená na Linuxovém jádře, odkoupena a následně vyvíjena společností Google, která slouží jako mobilní operační systém. Dnes se stala tato platforma základem pro mnohá další chytrá zařízení, jako jsou tablety, hodinky, televize a další [8].

1.1 Architektura operačního systému Android

Android OS je možné rozdělit do několika vrstev (Obr. 1). Každá vrstva plní danou úlohu a dohromady tvoří celek, ve kterém každá aplikace běží ve svém aplikačním sandboxu a může využívat software a hardware, který ji byl uživatelsky povolen [10].



Obrázek 1 Architektura Android OS [10]

1.1.1 Linuxové jádro

Petr Juhaňák [9] popisuje jádro Linuxu takto: „*Základním stavebním kamenem každého Androidu je linuxové jádro, které spolu s ovladači zařízení obsluhuje hardware telefonu. Linuxové jádro poskytuje elementární operace typu spustí aplikaci, čti/zapiš soubor nebo otevři síťové spojení.*

Linuxové jádro má kolem sebe další podpůrné prostředky a konfigurace, které jsou pro Androida specifické. Například při instalaci nové aplikace operační systém Androidu vytváří nového uživatele a stejnojmennou skupinu. Následně pak vytváří její interní datové úložiště v adresáři /data/data/jméno-aplikace a nastaví potřebná vlastnická práva.

(...)

Mobilní aplikace jsou spouštěny v tzv. "sandboxu", což znamená, že operační systém pro každou aplikaci vyhradí nový proces s virtuálním Java strojem (JVM), který teprve interpretuje programový kód mobilní aplikace.“ [9].

I přesto, že je operační systém Android založený na linuxovém jádře, nechová se jako klasický Unix-like systém, hlavním rozdílem je zde to, jak je pojatý multi-user přístup a sice, že každá aplikace je spouštěna jako samotný uživatel, čímž Android vytváří aplikační sandbox a aplikace jsou od sebe bezpečně odděleny [8].

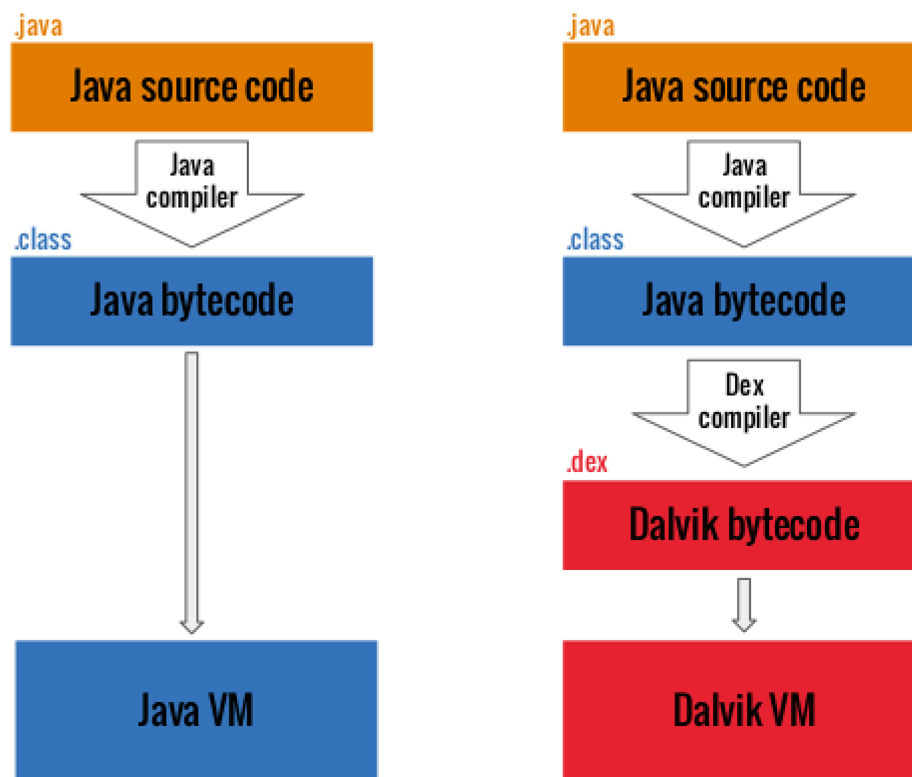
1.1.2 Abstraktní vrstva HAL

HAL je zkratkou pro Hardware abstract layer, poskytuje standardní rozhraní, pro práci s hardwarem. Abstraktní vrstva je tvořena sadou modulů, kde každý modul implementuje specifickou HW komponentu. Pokud si Java API framework vyžádá přístup například ke kameře, pak Android načte modul kamery (za předpokladu, že uživatel přijal příslušná oprávnění), který je napsán v C nebo C++ [8].

1.1.3 Běhové prostředí ART (Dalvik Virtual Machine)

ART, tedy běhové prostředí operačního systému Android – původně Dalvik Virtual Machine. Na rozdíl od klasické Javy kód zkompilovaný do bytecodu Javy je znova kompilován a vytváří spustitelný soubor *.dex (Obr. 2) [9]. Od Android 5.0 (API level 21) je výchozí běhové prostředí ART. Android Runtime je nástupcem původního běhového prostředí, které využívá dopředné kompilace, která se provádí pouze jednou. ART je navrženo pro potřeby operačního systému Android, kde každá aplikace běží ve svém

virtuálním zařízení a snaží se minimalizovat paměťové nároky, například použitím *.dex souborů, které jsou vytvořeny speciálně pro Android OS [10].



Obrázek 2 Java vs Dalvik [9]

1.1.4 Nativní knihovny C/C++

Vzhledem k tomu, že mobilní aplikace potřebuje vyžívat i různé komponenty, které samotná Java nemá implementovány, tak Java API framework nabízí možnosti, jak z Javy využívat nativní knihovny psané v programovacím jazyce C/C++. Příkladem může být třeba kamera, se kterou se dá pracovat pomocí Camera API, dostupné z Java API frameworku [10].

1.1.5 Aplikační framework Java API

Java API framework je základní stavební kámen, který vývojářům poskytuje funkce potřebné k běhu aplikace. Lze si jej představit jako sadu knihoven, se kterými může programátor pracovat. Obecně se snaží zjednodušit vývoj a poskytnout všechny potřebné funkcionality. Například při vývoji nemusí vývojář řešit, jaký typ HW má dané zařízení a psát k němu ovladače, ale jen pomocí příslušné API implementuje potřebnou aplikační logiku, navíc je framework optimalizován přímo pro potřeby operačního systému Android.

Zajímavostí je, že programátoři mají plný přístup ke kompletnímu frameworku, stejně jako systémové aplikace [10].

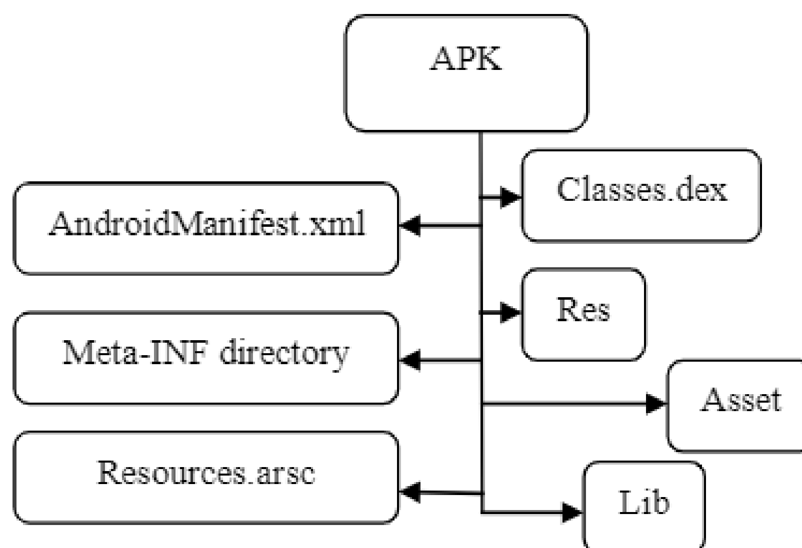
1.1.6 Systémové a uživatelské aplikace

Poslední část Android OS tvoří již samotné aplikace. Systémové aplikace jsou takové, které jsou dodávány výrobcem a nelze je přímo odinstalovat, jelikož k tomu uživatel nemá přístup. To může znamenat pro zařízení velké bezpečnostní riziko za předpokladu, že systémová aplikace obsahuje nějakou zranitelnost vůči malware. Uživatel se nemůže bránit, jelikož nemá přístup ke smazání systémové aplikace. Tyto aplikace však nemají žádný speciální status (existují výjimky – například nastavení), takže aplikace pro posílání SMS zpráv může být nahrazena aplikací třetí strany. Uživatelské aplikace jsou dostupné prostřednictvím distribuční platformy Google Play nebo internetu (například obchody třetích stran) [10].

1.2 Aplikace běžící v Android OS

Struktura mobilní aplikace představuje stěžejní bod pro bezpečností analýzu, proto je třeba ji detailně porozumět. Aplikace pro operační systém Android je tvořena APK balíčkem, který je využíván i jako instalační soubor. Součástí APK balíčku je také soubor AndroidManifest.xml, který je popsán v následujícím oddíle (Obr. 3). Mobilní aplikace pro Android OS jsou psány v jazyce Java, nebo Kotlin [11], [12].

1.2.1 Struktura aplikace



Obrázek 3 Struktura APK [12]

1.2.1.1 Soubory

AndroidManifest.xml je soubor typu .xml obsahující základní informace o aplikaci - například název balíčku, ikonu a verzi pro kterou je určen. Manifest je vstupním bodem aplikace, ale i hlavním zdrojem informací o aplikaci. Programátor, nebo tvůrce mobilního malware zde žádá o přístup k určitým chráněným částem Java API frameworku pomocí oprávnění, takže zde musí přiznat s jakými částmi API bude pracovat – to poskytuje představu o záměru aplikace. Pokud tak vývojář neučiní je vyvolána tzv. SecurityException, tedy výjimka, která je vyvolána při narušení bezpečnosti. Také zde vývojář informuje o službách a BroadcastReceiverech, které aplikace využívá (popsáno v sekci 1.2.2). Tento soubor musí mít každá mobilní aplikace pro operační systém Android. Výše uvedené skutečnosti naznačují, že AndroidManifest.xml je důležitý soubor, jak pro manuální statickou analýzu, tak pro automatizované antivirové skeny [11], [13].

Dle [11] lze manifest rozdělit do dvou hlavních analytických oblastí:

1. Oprávnění – pod tagem uses-permissions
2. Funkce HW nebo SW – pod tagem uses-features

```
▼<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.lemon.metamask"
 platformBuildVersionCode="11" platformBuildVersionName="1.1">
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
 ▼<application android:allowBackup="true" android:icon="@mipmap/ic_launcher" android:label="@string/app_name"
  android:roundIcon="@mipmap/ic_launcher" android:supportsRtl="true" android:theme="@style/AppTheme">
  ▼<activity android:name="com.lemon.metamask.Activity.MainActivity" android:theme="@style/AppTheme.NoActionBar">
   ▼<intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
   </activity>
   <activity android:name="com.lemon.metamask.Activity.CreateActivity" android:theme="@style/AppTheme.NoActionBar"/>
   <activity android:name="com.lemon.metamask.Activity.WalletSeedActivity" android:theme="@style/AppTheme.NoActionBar"/>
   <activity android:name="com.lemon.metamask.Activity.RestoreActivity" android:theme="@style/AppTheme.NoActionBar"/>
   <activity android:name="com.lemon.metamask.Activity.PrivateKeyActivity" android:theme="@style/AppTheme.NoActionBar"/>
   <meta-data android:name="com.android.vending.derived.apk.id" android:value="1"/>
  </application>
</manifest>
```

Obrázek 4 Ukázka AndroidManifest.xml [zdroj vlastní]

classes.dex je spustitelný soubor Dalvik, který má vždy stejné jméno – classes.dex. Soubor obsahuje všechny Java třídy zkompilovány právě do tohoto souboru, to znamená, že je v něm zahrnuta veškerá aplikační logika. Při analýze slouží tento soubor k získání zdrojového kódu [15].

resources.arsc je binární soubor vytvořený kompilací, který obsahuje jednoduché hodnoty typu integer, boolean, string, nebo i složitější UML layouty, které jsou ovšem v separátních XML souborech [16].

1.2.1.2 Adresáře

Adresář META-INF obsahuje podpisové soubory CERT.SF a CERT.RSA a soubor manifestu MANIFEST.MF [14].

Adresář Assets může obsahovat soubory, se kterými bude aplikace pracovat – binární, .xml, .db a další – k načítání se využívá Asset Manager z Java API Frameworku [17].

Adresář Lib obsahuje kód, který je specifický pro procesor [14].

1.2.2 Hlavní komponenty aplikace

Aktivita je základním prvkem všech aplikací. Jedna aktivita představuje aplikační logiku jednoho displeje, se kterou může interagovat uživatel. Na rozdíl od ostatních programovacích paradigmat zde nefiguruje spouštěcí metoda *main()*, ale využívá se instance třídy Activity. Pakliže jsou v aplikaci 4 obrazovky, jsou v ní i 4 aktivity, tedy celkem 4 instance třídy Activity. Všechny aktivity musí být deklarovány v souboru AndroidManifest.xml pomocí activity tagu, jinak je při načtení vyvolána výjimka [1], [18].

```
<activity android:name="com.lemon.metamask.Activity.WalletSeedActivity"
android:theme="@style/AppTheme.NoActionBar"/>
```

Obrázek 5 Deklarace aktivity [zdroj vlastní]

Životní cyklus aktivity: Životní cyklus aktivity je tvořen stavy, mezi kterými lze přecházet pomocí tzv. callbacků (Obr. 6). Ve výchozím stavu je nutné implementovat pouze metodu *onCreate()*, která je volána při spuštění aktivity. Jednotlivé callback funkce je možné přepsat (@Override) a implementovat dle potřeb aplikace [1], [19].

Pomocí callbacků aktivita rozpozná, jestli je právě vytvořena, zastavena nebo přerušena. Callback umožňuje vývojářům implementovat, jak se bude aktivita právě v těchto různých stavech chovat. Zajímavostí je zde zejména časté volání callbacku *onResume()* [19]. Obrázek 6 naznačuje, jak často je tato metoda volána – tedy vždy, když má aktivita být přesunuta do popředí.

Obrázek 6 Životní cyklus aktivity [19]

Oprávnění (Permissions) by se dala představit, jako klíče k daným částem Java API frameworku. Pokud uživatel povolí aplikaci využívat externí datové úložiště, pak aplikaci dovolí pracovat s částmi Java API frameworku pro externí datové úložiště (android.permission.WRITE_EXTERNAL_STORAGE). Oprávnění se deklarují v AndroidManifest.xml pomocí uses-permission tagu. V původním binárním modelu uživatel buď přijal všechna oprávnění a aplikace byla nainstalována, nebo oprávnění nepřijal a aplikaci si nenainstaloval. Od Android 6.0 (API level 23 a vyšší) je využíván nový model oprávnění, který uživateli umožňuje spravovat oprávnění aplikace za běhu a umožňuje tak akceptovat, nebo zamítnout jednotlivá oprávnění pro nainstalované aplikace [20]. Dle [11] se jako doplněk při detekci k oprávněním dá využít uses-feature tag, jehož záměrem je informovat o HW a SW funkcích, na kterých aplikace závisí.

Rozdělení dle úrovně ochrany podle [21]:

A. Normal permissions: normální oprávnění se vztahují na oblasti, které nepředstavují bezpečnostní riziko pro uživatele. Tato oprávnění jsou uděleny při instalaci a uživatel je nemůže zrušit – takovým oprávněním je např. přístup k internetu, nebo nastavení časové zóny.

B. Dangerous permissions: nebezpečná oprávnění se vztahují na oblasti, které naopak mohou představovat pro uživatele bezpečnostní riziko – uživatel buď aplikaci schválí přístup, nebo mu nebude poskytnuta funkcionalita, která dané oprávnění využívá – například číst kontakty uživatele, nebo využívat kameru.

C. Signature permissions: oprávnění dle certifikátu je uděleno systémem aplikaci při instalaci. Ale pouze za předpokladu, že aplikace, která žádá toto oprávnění je podepsána stejným certifikátem jako aplikace, která toto oprávnění definuje.

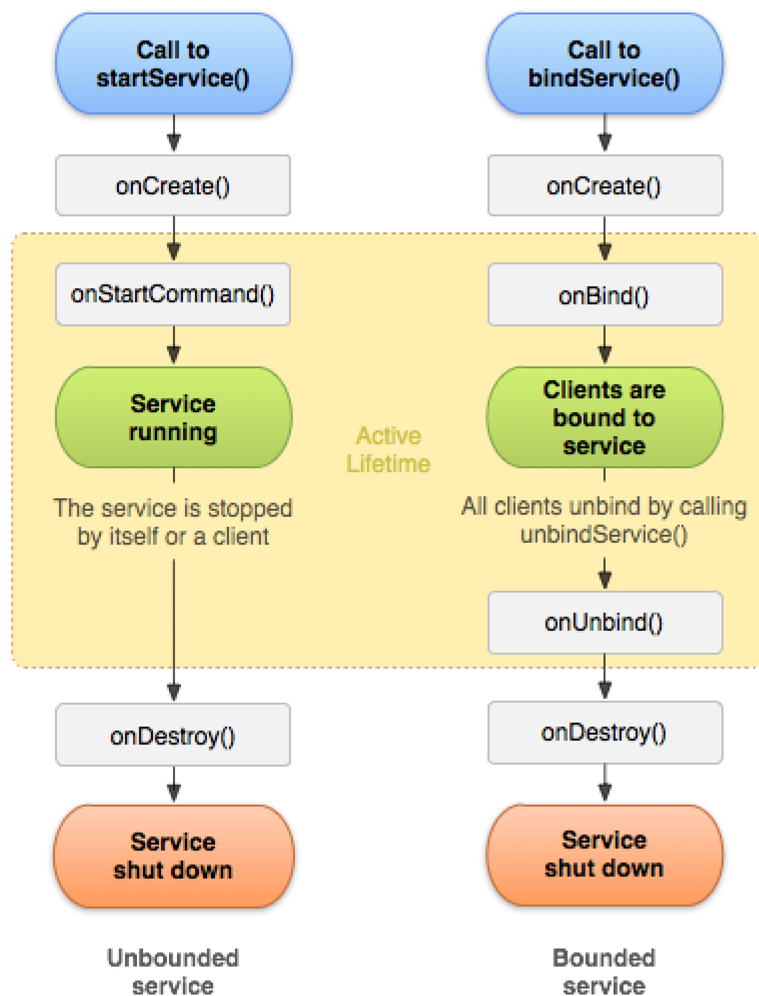
D. Special permission: tato speciální oprávnění by měly aplikace využívat co nejméně, jelikož pracují s citlivými daty, například umožňují aplikaci číst, nebo měnit nastavení telefonu. Tato oprávnění se nechovají ani jako normální, ani jako nebezpečná oprávnění. Pokud chce aplikace využít speciální oprávnění je nutné jej deklarovat v manifestu a odeslat intent vyžadující autorizaci uživatele. Systém odpovídá zobrazením obrazovky s detailními informacemi.

Autoři [B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas & G. Á. Marañón, 11] však ve své práci naznačují, že normální oprávnění nemusí nutně znamenat, že jde o bezpečnou aplikaci. V jejich sadě vzorků bylo oprávnění s nejvyšší absolutní četností z kategorie normálních a sice android.permission.INTERNET. Přesto, že se jedná o normální oprávnění, analýzy provedené v [11] naznačují jeho intenzivní zneužívání mobilním malwarem. Oprávnění bylo například využíváno pro odesílání a příjmem textových zpráv.

Služby (services) jsou určeny pro vykonávání dlouhodobě běžících procesů, přičemž musí být deklarovány v souboru AndroidManifest.xml pomocí tagu service. Service je komponenta aplikace, která neposkytuje uživatelské rozhraní, ale pracuje na pozadí – jejich systémová priorita je nižší než u aktivní aplikace, avšak vyšší než u neaktivní aplikace, což znamená, že nejsou tak často ukončovány z důvodů zatížení systému [1], [22].

Z pohledu životního cyklu jsou dle [22] rozděleny služby na vázané a nevázané (Obr. 7). Na rozdíl od vázaných služeb, které zanikají spolu s vázanou komponentou (např. aktivita) je

vývojář povinen nevázané služby ukončit vlastní implementací, pokud tak neučiní service může běžet na pozadí nekonečně dlouho, nebo do zastavení systémem. Při použití nevázané služby je nutné zajistit její zastavení a to buď pomocí metody *stopSelf()* (služba ukončuje sama sebe), nebo pomocí *stopService()* (ukončení z jiné komponenty).



Obrázek 7 Životní cyklus služeb [22]

Content Providers slouží k poskytování dat, například pro přenos dat mezi aplikacemi a také při externímu přístupu k datům. Využívají REST přístupu a URI adresovacího schématu, kde všechny poskytovatelé mají model: *content://*. Jedná se o mechanismus, ve kterém aplikace prostřednictvím Content Provideru poskytuje svá data ostatním programům [23].

Aplikace, které chtějí přijímat data musí mít implementován Content Resolver. Pokud není potřebné oprávnění k provideru, pak jej může použít jakákoliv aplikace, proto by měli vývojáři specifikovat příslušná oprávnění, čímž lze zabránit nechtěným únikům dat [23], [25].

Broadcast Receiver slouží k posílání, nebo přijímání zpráv od operačního systému Android, nebo jiných aplikací. Umožňuje přihlásit receiver k určitým událostem systému, nebo aplikací. Jakmile nastane určitá událost, např. připojení telefonu k nabíječce jsou všechny přihlášené receivery informovány o tom, že událost proběhla. V Broadcast Receiveru je následně zavolána metoda *onReceive(),* jejíž kód je vykonán [24].

Jsou využívány zejména k obsluze, nebo reakci na určitou událost. Při používání receiverů hrozí nebezpečí, že některá aplikace bude broadcasty zneužívat – například: zařízení se připojilo k Wi-Fi, takže malware může začít odesílat odcizená data, bez čerpání celulárního datového tarifu. Čerpání datového tarifu by mohlo vést ke kompromitaci malwaru [24], [25].

Broadcast receiver může být deklarován v manifestu, ale i takzvaně „Context-registered", to znamená, že je deklarován využitím metody *registerReceiver(BroadcastReceiver, IntentFilter)*. Rozdíl mezi těmito způsoby je v tom, že při použití deklarace v manifestu může být receiver zavolán i pokud aplikace právě neběží. Pokud broadcast obsahuje citlivá data, pak by měl vyžadovat i příslušná oprávnění, aby se k nim nedostaly škodlivé aplikace [24].

Třída LocalBroadcastManager pomáhá vývojářům, s registrací a odesíláním lokálních broadcastů, které zůstávají v aplikaci. Programátor nemusí řešit případný únik dat [24], [25]. Dle [1] Google však označil ve své dokumentaci třídu LocalBroadcastManager za zastaralou a doporučuje využívání alternativ – například LiveData [1].

Jako další bude popsán tzv. Intent. O IPC, tedy komunikaci mezi jednotlivými procesy se stará Binder, což je vlastní implementace IPC mechanismu postavená na OpenBinder technologii. Intent je založen právě na Binderu, to napovídá, že intent je komunikační framework, který pomocí zpráv umožňuje komunikaci mezi jednotlivými komponentami aplikace. Nejčastější využití nachází tento framework, pokud uživatel při používání aplikace přechází z jedné aktivity na jinou. Vývojář použije intent s příslušnými parametry a uživatel je přesměrován na požadovanou aktivitu. Intent je využíván i pro přenos dat mezi komponenty aplikace. Intent je složen ze dvou hlavních částí, jednak action – popisuje operaci, která bude vykonána a jednak data – tedy data, se kterými bude operace pracovat.

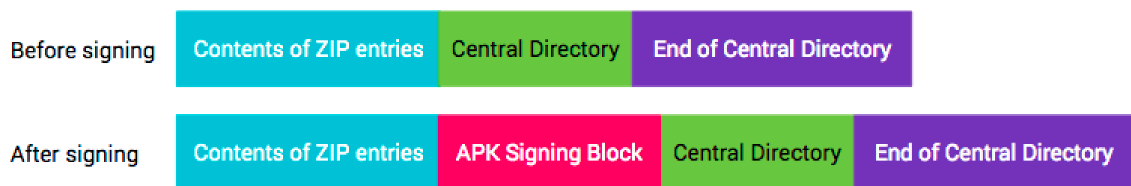
Intent je možné použít ke spuštění aktivity jeho předáním, jako parametru do metody *startActivity(Intent)*. Intent lze rovněž využít ke spuštění služby na pozadí, předáním intentu, jako parametru metodě *startService(Intent)*. Také jej lze využít pro odesílání broadcastu, který může přijmout jakákoliv aplikace předáním intentu jako parametru do metody *sendBroadcast(Intent)*. Explicitní deklarace se využívá především při spuštění komponent a přesně specifikuje komponentu plným názvem její třídy. Implicitní deklarace se například využívá, pokud chce vývojář ukázat uživateli polohu na mapě – intent je odeslán operačnímu systému Android a uživatel, ani vývojář již neovlivní to, jakou aplikaci, nebo třídu k tomu systém určí [1], [26], [27].

Dle [26] lze intent chápat, jako abstraktní popis nějaké operace, která má být provedena. Z výsledků publikovaných ve [27] plyne, že intent může poskytnout i velmi zajímavý ukazatel, co se týče problematiky škodlivých aplikací. Z celé datové sady (7 406 aplikací) bylo dosaženo 91% úspěšnosti při detekci malware na základě intentů. Navíc tento experiment ukázal, že kombinací detekce oprávnění a intentu lze dosáhnout úspěšnosti detekce malwaru 95,5 % [27].

1.2.3 Podepisování aplikace

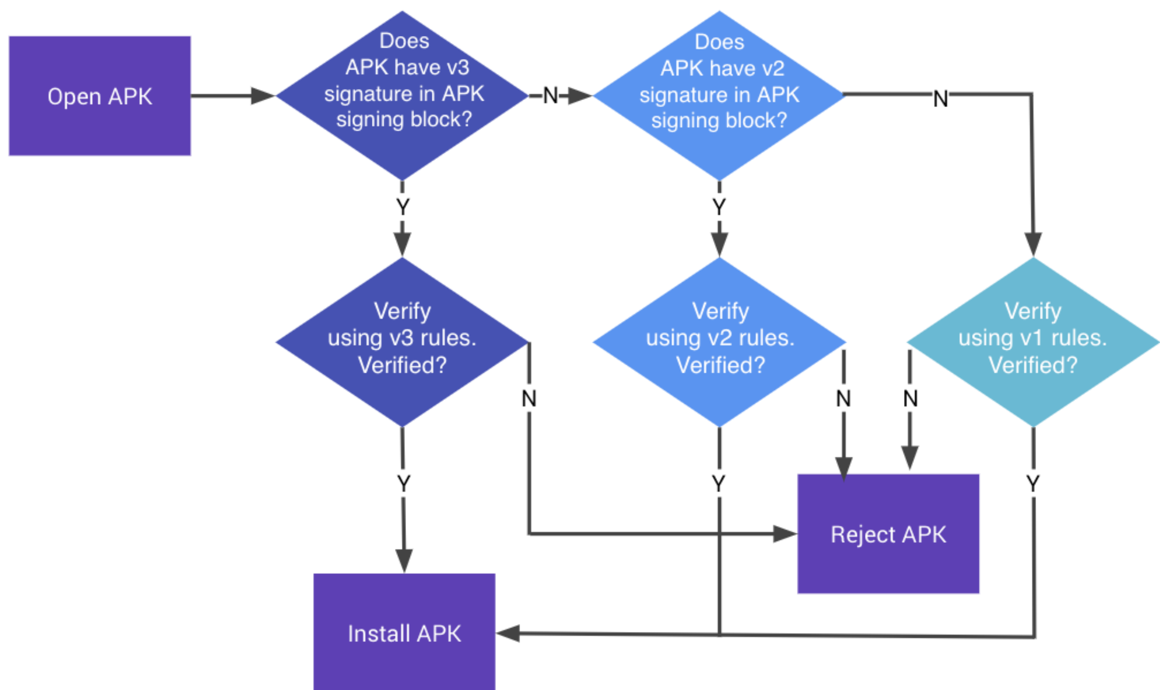
Každá aplikace spouštěná na platformě Android musí být od vzniku operačního systému Android podepsána vývojářem. Pokud aplikace nejsou podepsány, pak jsou zamítnuty buď distribuční platformou Google Play, nebo Package Installerem, tedy objektem sloužícím k instalaci aplikací pomocí APK souboru přímo na zařízení. Package Installer mimo jiné ověřuje certifikáty a dokáže rozpoznat stejnou vývojářskou společnost. Díky tomu mohou aplikace stejného dodavatele sdílet uživatelské id, což umožňuje sdílení dat v privátním prostoru data/data. Nejedná se však o běžné certifikáty, které se generují při normálním vývoji aplikace a slouží především k ladění programů [1]. Vzhledem k tomu, že většina důvěryhodných obchodů s aplikacemi vyžaduje pro reálnou publikaci bezpečný certifikát, je nutné vývojáře ověřit pomocí veřejného klíče, jenž je svázán s aplikací. Android OS nevyužívá stejný systém podepisování aplikací od svého počátku, proto existují tři systémy podepisování. Původním systémem je podepisování JAR (označováno v1 Scheme). Systém podepisování JAR musel zahrnovat všechny položky souboru MANIFEST.MF podepsané společným klíčem, ale byl hodně paměťově a výpočetně náročný. Zpracovával obsáhlé datové struktury. Navíc při ověřování bylo nutné dekomprimovat všechny komprimované soubory. Tento systém podepisování nechránil všechny části APK souboru, například metadata. Proto s příchodem Androidu 7.0 (API level 24) byl vytvořen nový systém

podepisování nazvaný APK Signature Scheme, který se značí jako Scheme v2. Hlavním rozdílem mezi první a druhou verzí podepisování je vložení takzvaného Signing blocku (Obr. 8). Tento podepisovací blok obsahuje tzv. párové hodnoty (název-hodnota) s informacemi a je do APK balíčku ještě před centrální soubor ZIP. Scheme v2 je navíc tzv. whole-file, což znamená, že se nezaměřuje pouze na část aplikace, ale na software jako celek. Tyto skutečnosti znamenají zrychlení ověřovacího procesu a zesílení integrity aplikací [1], [27].



Obrázek 8 APK soubor před a po vložení podepisovacího bloku [27]

Od Android OS verze 9.0 (API level 28) a výše je dostupný třetí a zatím poslední systém APK Signature Scheme V3. Toto schéma kromě informací o SDK přidává hlavně proof-of-rotation funkci, která je určena spíše na vývojáře. Ve zkratce se jedná o to, že existuje propojený seznam, ve kterém každý prvek představuje certifikát jedné verze aplikace. V tomto seznamu každý předek potvrzuje platnost svého potomka, tím každý nový potomek reprezentuje novou identitu, která může podepsat APK. To aplikaci umožňuje měnit její podepisovací klíč, jako část svého update. Zmiňované systémy jsou samozřejmě zpětně kompatibilní. Nejprve se otestuje, jestli je APK podepsané dle systému Scheme V3. Pokud má podpis typu V3, APK je buď nainstalováno, nebo odmítnuto uživatelem. Pokud podpis typu V3 nemá, otestuje se jedná o typ V2. Analogický postup se aplikuje až po podpis typu V1 (Obr. 9) [1], [27], [28].



Obrázek 9 Validační proces [1]

2 MALWARE A STATICKÁ ANALÝZA

Kvůli obrovskému nárůstu a vyspělému chování, může malware působit jako relativně nový bezpečnostní problém. Uživatelé se mohou setkat například i s umělou inteligencí. Skutečnost je však taková, že škodlivé softwary existují už od 60. let minulého století. Například hra, která se jmenovala Darwin, ve které dva hráči soupeřili o to, kdo dříve vymaže všechny programy protihráče. Brzy si však vědci a softwaroví inženýři uvědomili, že tato původně neškodná hra může být velmi nebezpečná, pokud by byla použita k jiným účelům. K rozmachu malwaru přispívá i to, že moderní zařízení nejsou tak finančně náročné. Jsou tedy mnohem dostupnější, na rozdíl od starých technologií, které měly velké rozměry a byly drahé. Tento fakt naznačuje, že s rostoucím počtem uživatelů roste i počet lidí, kteří objevují jejich slabiny a zranitelnosti [29].

2.1 Malware

Pojmy malware a virus bývají velmi často chybně zaměňovány. Často bývají škodlivé programy nazývány jako viry, nicméně vir je neodborný název, který se používal v minulosti jako obecný pojem pro malware. Slovo malware vzniklo spojením dvou anglických slov a sice malicious a software, tedy škodlivý software. První malwary si kladly za cíl uživatele pouze vystrašit tím, že se jejich počítač začal chovat jinak, než očekávali [29]. Postupem času se škodlivé programy objevovaly čím dál častěji a začaly představovat větší riziko. V 90. letech minulého století se velmi často používaly škodlivé softwary pro krádeže citlivých informací, jako jsou například bankovní účty a hesla [29]. Je dokázáno, že malware je využíván i v armádních sférách. Například Department of Homeland Security Spojených států amerických [30] na svých webových stránkách uvádí více, než 20 útoků které provedla Severní Korea. Útoky se zaměřovaly především na krádeže informací a peněz [29], [31], [32].

2.1.1 Distribuce malwaru

Prvním krokem v životním cyklu malwaru je jeho distribuce do hostitelských zařízení. Malware je možné stáhnout jednak z oficiální distribuční platformy Google Play a jednak z neoficiálních zdrojů – obchody třetích stran, torrenty a file share servery. Malware se snaží nalákat uživatele, aby si stáhl a nainstaloval škodlivou aplikaci, aniž by tušil, že stahuje malware. Autoři Jiang Xuxian a Yajin Zhou [33] na základě svého výzkumu kategorizovali

distribuční techniky do tří základních kategorií a sice *repackaging*, *update attack*, a *drive-by download* [33].

2.1.1.1 Repackaging

Jedná se o takzvané „přebalování“ aplikace, princip spočívá v tom, že si tvůrce mobilního malwaru vybere nějakou legitimní aplikaci z obchodu, typicky jde o populární programy, které mají vyšší počet stažení. Malware se často vydává za neplacené verze placených softwarů (například hry, ale i pornografické aplikace). Programátor mobilního malwaru tuto aplikaci dekompiluje, například pomocí nástroje Apktool. Následně do programu přidá škodlivou funkcionalitu a provede opětovnou kompilaci. Aplikaci s touto funkcionalitou zveřejní v některém oficiálním (Google Play) nebo neoficiálním obchodě (třetí strany). Infikované aplikace lze šířit i jiným způsobem, například prostřednictvím file share serverů. Tento útok je založen na manipulaci se zdrojovým kódem a k jeho odhalení slouží statická analýza, která se nezabývá chováním aplikace, ale zdrojovým kódem [33]. Výzkum popsaný v [33] ukázal, že se jedná o velmi používanou metodu, jelikož v datové sadě, čítající 1 260 vzorků malwaru, bylo 1 083 vzorků malwaru testováno pozitivně na repackaging, což je 86,0 % z celé datové sady.

2.1.1.2 Update Attack

Tato metoda nabízí tvůrcům mobilního malwaru velmi sofistikovaný přístup, díky kterému je možné obcházet detekci založenou na signaturách. Změny v softwaru se dějí za běhu. Jednou z možností je, že se škodlivé komponenty načítají ze zdrojů aplikace, například ze souborů JAR, takový typ změny nevyžaduje ani povolení uživatele a je velmi složité ho detekovat, jelikož nestahuje žádnou další verzi aplikace, ale jen upravuje své části. Další možností je nabídnutí uživateli pomocí notifikace, nebo dialogového okna novou verzi aplikace. Tato aplikace může být načtena přímo z hostitelské aplikace (jako resource soubor, nebo asset), ale také může být stažena vzdáleně z internetu, takže nemusí být součástí dané aplikace. Obecně lze o technice update attack uvažovat jako o „trojském koni“, který se vydává za důvěryhodnou aplikaci, jenž za běhu mění svou funkcionalitu a stává se škodlivou. Tento proces velmi komplikuje celou detekci malwaru, jelikož k jeho detekci je ideální dynamická analýza, která může aplikaci sledovat za běhu [33], [34].

2.1.1.3 Drive-by download

Technika drive-by download se objevuje, jak na mobilních zařízeních, tak i na počítačích. Pracuje stylem „drive-by“ – „projetím kolem“. Ve zkratce to znamená, že stačí pouhé otevření infikované webové stránky k zahájení stahování škodlivého softwaru. Často se využívá takzvaných „exploit kitů“, které tvůrci mobilního malwaru používají k vytipování zranitelných míst na webu, nebo v zařízení. Tento útok často pramení v nedostatečné aktualizaci zařízení, což může znamenat bezpečnostním riziko pro uživatele. Aktualizace jsou důležité, protože nalezení zranitelnosti, které vývojář malwaru může potencionálně využít, bývá opraveno právě v aktualizacích operačního systému. Drive-by download se obvykle šíří pomocí nakažených webových stránek a nemusí to být žádné komplexní a rozsáhlé programy, právě naopak jsou to jen malé části kódu, které si za svůj cíl kladou spojit se s dalším zařízením, ze kterého získají další potřebné části kódu [35].

2.1.2 Využití malware

Autoři Jiang Xuxian a Yajin Zhou [33] ve své práci použili základní rozdělení dle zaměření malwaru. Jelikož druhů malwaru existuje velké množství rozdělili datovou sadu do čtyř kategorií podle toho, jakých funkcionalit využívají a jaké je jejich zaměření. Tyto kategorie jsou *privilege escalation* (využívá chyb anebo nedostatků programu), *financial charge* (finančně motivované), *remote control* (cílem je získat vzdálený přístup k zařízení), *information collection* (sběr dat ze zařízení) [33].

2.1.2.1 Privilege escalation

Za elevaci oprávnění – tedy privilege escalation je obecně považován stav získání více schopností, než by mělo být správně uděleno. Operační systém Android je open-source software s více než 90 open-source knihovnami, což znamená, že je velmi komplexní. Následkem je vyšší množství potencionálních zranitelností, které jsou ideálním cílem útočníka mířícího na elevaci oprávnění [33].

Systém oprávnění na platformě Android uvažuje o aplikacích pouze jednotlivě a odděleně, nebere v úvahu IPC komunikaci a možnost složení aplikací, případně oprávnění. Tvůrce mobilního malwaru tak může využít tohoto mechanismu a pomocí kombinace oprávnění aplikací tak zvýšit oprávnění své aplikace. Zároveň není možné chtít po každém vývojáři, aby byl schopen předejít tomu, že jeho aplikace bude zneužita ke zvýšení oprávnění [36].

2.1.2.2 Financial Charge

Druhou kategorii tvoří finanční poplatky – tedy financial charges. Obecně se jedná o funkcionalitu, která má za následek finanční poškození uživatele. Typicky se jedná o přihlášení odběru prémiové služby (placené), ale nemusí to tak být vždy, může se jednat o placené hovory na pozadí. V ideálním případě lze tyto útoky odhalit statickou analýzou, aplikace posílající SMS musí totiž mít oprávnění *android.permission.SEND_SMS*, které udělí přistup k metodě *sendTextMessagge()*, ta umožňuje odesílat SMS na nějaké telefonní číslo. Dle oprávnění, případně podle metody je možné malware odhalit statickou analýzou. Některé programy však nemají tato čísla hard-coded – napsané přímo v kódu, ale využívají metody remote control a stahují si je za běhu [33], [37].

2.1.2.3 Remote Control

Kategorie vzdáleného přístupu – tedy remote control – se snaží získat kontrolu nad zařízením tím, že komunikuje s C&C (command-and-control) serverem. Server posílá příkazy, které má zařízení vykonat, nebo je využíván jako místo, kde hlásí infikovaná zařízení výsledky, případně odesílají ukradená data. Tato technologie je často využívána k tvorbě botnetu [38].

V práci [33] byla provedena analýza, která naznačuje, z celkového počtu 1 260 vzorků bylo 93,0 % pozitivně testováno na komunikaci se vzdálenými servery, nebo na zneužití zařízení k vytvoření bota. Autoři v [33] uvádějí, že některé malwary šifrují URL adresy i komunikaci, čímž se snaží působit nenápadněji před anti-malwarovou ochranou. Při hledání určité URL adresy potom ochranný software nemusí brát v úvahu šifru, možným řešením pro detekci by poté mohla být analýza na základě sekvence volání JAVA API frameworku [33].

2.1.2.4 Information collection

Poslední, avšak neméně důležitou kategorii tvoří sběr informací – tedy information collection. Obecně se jedná o sbírání dat ze zařízení, tato data mohou být například SMS zprávy, kontakty, přihlašovací údaje, bankovní účty, poloha zařízení nebo třeba fotografie. Tyto údaje jsou pak odesílány ze zařízení, často na vzdálené servery. Pokud uživatel využívá napadené mobilní zařízení pro práci, pak mohou být odcizena i firemní data/know-how [39].

Často bývají tyto metody kombinovány. Například pomocí vzdáleného přístupu lze realizovat jak sběr informací a přihlášení k prémiovým službám, tak i elevaci oprávnění [33].

2.2 Statická analýza

Statická analýza kódu by měla doprovázet vývojáře při tvorbě jejich aplikací, nicméně zásadní je její využití při detekci malwaru. Techniky statické analýzy nevyžadují vykonávání kódu, namísto toho pouze analyzují kód, což má pozitivní vliv na výkon zařízení. Pomocí statické analýzy je možné detekovat statické příznaky malwaru – komponenty, oprávnění, sekvence volání frameworku a další příznaky nevyžadující spuštění aplikace. Statická analýza se hojně využívá k vytváření vzorů, které následně slouží pro porovnávání vzorů, tedy pattern matching. Na základě těchto vzorů probíhají detekce malwaru [36].

V práci [15] dělí autoři statickou analýzu na tři kategorie. Signature-related, tedy orientovanou na příznaky, tvoří první kategorii zaměřenou na sesbírání příznaků malwaru. V této kategorii se využívá vzorů v kódu, textových řetězců a dalších znaků, které jsou potom využity při klasifikaci malwaru. Druhou kategorii tvoří permissions-related, tedy zaměřenou na oprávnění aplikace, které získává ze souboru AndroidManifest.xml. Uvedená metoda často nezkoumá pouze permissions, ale i další informace z manifestu, například uses-feature tagy. Poslední kategorií je analýza classes.dex souborů. APK balíček se dekompiluje, což umožní práci se zdrojovým kódem [15].

Jiný pohled na věc poskytuje rozdělení na dvě kategorie, a sice signature-based a permission-based. Permission-based přístup je založen na oprávněních, který spoléhá na systém oprávnění, kdy aplikace pro určité funkcionality vyžaduje specifická oprávnění. Aplikace však může žádat oprávnění, která nepotřebuje, to může vést ke špatnému vyhodnocení malwaru, tzv. false positive, kdy je označena legitimní aplikace chybně jako malware. Druhý přístup patří mezi nejčastěji využívané přístupy v komerčních bezpečnostních řešeních a je označován jako signature-based. Tato technika je založena na předchozí znalosti malwaru, což je její hlavní nevýhodou. Aby správně fungovala musí být nejprve aplikace posouzena jako malware. Z nakažených aplikací je možné statickou analýzou získat hlavní znaky aplikace, které lze později použít pro detekci a klasifikaci malwaru [40].

2.2.1 Obfuskační techniky

Tyto techniky se dají považovat za největšího nepřítele statické analýzy a reverzního inženýrství. Vznikly aby znemožnily, nebo alespoň ztížily reverzní inženýrství. Obecně se snaží změnit velikost, nebo obsah souboru, aniž by se změnily jeho funkce [41]. Obfuskačních technik existuje několik, jedná se o: *namely identifier*, *string encryption* a *control flow obfuscation* [42].

2.2.1.1 Namely identifier

Technika, kterou vývojáři malwaru používají ke zhoršení čitelnosti kódu a změnění obsahu aplikace. Aby tak učinili, mění například názvy proměnných, ale i tříd. Existují i nástroje, které to dělají za ně a nazývají se obfuskátory. Ty fungují tak, že vezmou jméno proměnné, například „obsah“ a změní ji na „a“. Obfuskátory většinou tvoří názvy složené z jednoho až tří písmen dle abecedy, tedy: „a“, „aa“, „ab“ a další. Při statické analýze se zhoršuje čitelnost a změní se tím i hash aplikace [42], [43].

2.2.1.2 String encryption

V aplikaci a jejich textových řetězcích se mohou nacházet různá citlivá data, nebo třeba URL adresy, pomocí kterých lze identifikovat malware. Tato technika, tedy string encryption se zaměřuje na šifrování řetězců, což komplikuje statickou analýzu – v kódu nejsou přímo čitelné hodnoty řetězců, které by mohly vést k detekci malwaru. K šifrování se často používá AES a DES, ale je čistě na tvůrci malwaru, jaký způsob si zvolí [42]. Vývojáři mobilního malwaru zašli s touto metodou ještě dále. Vymysleli class encryption (šifrování třídy), čímž téměř znemožnili detekci na základě statické analýzy. Využívají totiž mechanismu, při kterém nejdříve zašifrují celou třídu, poté jí zkomprimují a uloží například jako asset. Pro tyto účely je vytvořena metoda, která se zavolá za běhu aplikace a provede opačný postup – dekomprimuje, dešifruje a načte celou třídu [41].

2.2.1.3 Control flow obfuscation

Princip této techniky spočívá ve změně CFG, tedy Control Flow Graph. CFG si lze představit jako diagram aplikace, který má větve a uzly. Cílem této techniky je diagram rozšířit – přidat větev/uzel, nebo zmenšit – odebrat větev/uzel. Toho lze docílit mnoha způsoby. Nejjednodušším způsobem pro rozšíření CFG je přidání smyčky, nevyužívaného kódu, redundantního kódu nebo reorganizace příkazů. Protikladem je odebrání, které funguje analogicky, jen se kód nepřidává, ale maže [42].

3 DETEKCE MALWARE

Uživatelé, jakožto koncoví zákazníci vyžadují od anti-malware programů zejména rychlost, proto kontrola v zařízení probíhá převážně na pozadí. Díky komerčním anti-virovým produktům stále přibývají různé techniky pro detekci malwaru [45].

3.1 Obecné rozdělení detekce malware

Detekce malwaru je možné rozdělit například dle toho, jestli se vykonávají u hostitele, tedy v mobilním zařízení, nebo na vzdáleném serveru. Dále se detekce dělí dle analýzy (statická, dynamická, hybridní). Nejdůležitější rozdělení je však dle detekčních technik. Hlavními třemi technikami pro detekci malwaru na platformě Android jsou *signature-based*, *heuristic-based* a *specification-based,* všechny tři techniky mohou být realizovány staticky, dynamicky i hybridně [46].

A. Signature-based: technika je také známá jako pattern matching. Využívá skutečnosti, že každý malware má nějaký znak, nebo znaky, podle kterých lze například identifikovat do jaké rodiny patří. Z těchto znaků pak vznikne vzor, který se uloží do databáze a následně se využívá k porovnávání. Z toho vyplývá, že aby byl malware na zařízení identifikován musí být uložen v databázi, tento systém detekce selhává v případě nových neznámých malwarů. Pokud je nalezena shoda mezi vytvořeným vzorem a aplikací, je aplikace klasifikována jako malware [47]. Na platformě Android komerční anti-virové společnosti například využívají hash aplikace. Hash aplikace není úplně ideálním bezpečnostním řešením. Díky obfuskačním technikám může být hash jednoduše změněna, proto například systém DroidAnalytics používá třístupňové signatury, kde sledují nejen celou aplikaci, ale i její třídy a metody [48].

B. Heuristic-based: technika někdy označovaná jako anomaly-based. Na rozdíl od signature-based techniky umožňuje detekovat i malware, který ještě nebyl analyzován, což s sebou nese riziko falešného poplachu u legitimní aplikace. Monitoruje se pravidelné, normální chování a chování škodlivé. Hledají se známky nějaké změny od vzoru normálního stavu, pokud je nalezen nějaký příznak podezřelého chování shodný se vzorem útoku, pak je aplikace označena jako potenciální malware. Tato metoda je náročnější na zdroje zařízení [47].

C. Specification-based: tato technika je odvozená od heuristické detekce, ale spíše než na vzory malwaru se zaměřuje na odchylky od normálního chování, které odpovídá

specifikaci dané aplikace. Tato technika se liší od heuristické také tím, že se u ní nevyskytují tak časté falešné poplachy, ať už pozitivní, nebo negativní [47].

3.2 Rozdělní dle zkoumaných dat

3.2.1 Permission-based

Vzhledem k tomu, že k daným metodám Java API frameworku lze přistoupit pouze na základě oprávnění, lze díky použitým oprávněním odvodit využití aplikace. Na uvedené skutečnosti je založen detekční systém, který provádí analýzu oprávnění. Tento systém pracuje s nainstalovanými a odinstalovanými aplikacemi. Využívá několik modulů, jeden je založen na dekompilaci pomocí nástroje Apktool a otevření souboru AndroidManifest.xml, ve kterém jsou oprávnění. Druhý modul využívá JAVA API frameworku, přesněji metodu *getInstalledPackages()*. Následně z instancí aplikací extrahuje oprávnění. Další modul ukazuje všechny nainstalované aplikace a detailní oprávnění všech aplikací. Poslední modul je seznam citlivých oprávnění a seznam všech aplikací, které jej využívají [49].

3.2.2 AndroidManifest-based

Vzhledem k tomu, že soubor AndroidManifest.xml musí mít každá aplikace, je ideálním místem pro detekci malwaru [50]. Příkladem detekčního systému založeného na analýze manifestu může být práce [50]. Metoda pro detekci malwaru je rozdělena do tří kroků. Prvním krokem je extrakce informací z manifestu. Je vytvořen seznam informací obsažených v infikovaných a v legitimních aplikacích. Z tohoto je vytvořen seznam klíčových slov pro detekci. Poté se pro jednotlivé informace v detekovaném programu (oprávnění, intent filter, název procesu) spočítá míra škodlivosti dána vztahem: $P = \frac{M-B}{E}$, kde P pravděpodobnost toho, že se jedná o malware; M je počet škodlivých informací, B je počet neškodných informací; E je celkový počet informací. V posledním kroku se využívá strojového učení a na základě podmínek se počítá konečné skóre, čím více se bude P blížit k 1, tím je pravděpodobnější, že se jedná o malware [50].

3.2.3 API-based

Volání API frameworku se v Android OS využívá nejen pro poskytování funkcionality vývojářům, ale i pro detekci malwaru. První pokusy se realizovaly pomocí logování a sloužily pro zmapování volaných funkcí. Následně začaly vznikat nástroje fungující na různých principech, některé kombinovaly permission-based a API-based [51]. Například

autoři Q. Li a X. Li [52] ve své práci používají nástroj, který nazvali DexParse.exe, sloužící pro extrakci dat o volání API frameworku. Nástroj je napsaný v programovacím jazyce C++ a je vytvořený tak, aby vracel několik souborů s ohledem na zaměření. Například soubor Classes.output vrací třídu s její rodičovskou třídou, ale používá i číselné ID, aby předešli problémům s obfuskací. Ze všech výstupů se vytvoří stromová charakteristika volání pro všechny třídy. Z této struktury se následně vypočítají podobnosti mezi škodlivým programem a právě detekovanou aplikací [52].

3.2.4 Certification-based

Z certifikátu lze extrahovat sériové číslo, což může být také jedním z vodítek při detekci malwaru. Funguje velmi jednoduše na základě blacklistu sériových čísel, což je seznam těch čísel, které byly nalezeny ve škodlivých aplikacích. Dosahuje velmi přesvědčivých výsledku, jelikož se často tato čísla u malwaru opakují. Pro minimalizaci falešných poplachů a zároveň pro docílení maximální flexibility detekce je ideální detekční metody kombinovat. [53].

II. PRAKTICKÁ ČÁST

4 STATICKÁ ANALÝZA VZORKŮ MALWARE

Statická analýza probíhala manuálně na aplikacích, které byly obsaženy v datové sadě. V první fázi analýza sloužila k výběru ideálních analytických nástrojů. Druhá část analýzy se soustředila na informace, které lze zjistit z testovaných aplikací. Zkoumaným objektem statické analýzy je soubor APK, ve kterém jsou aplikace pro Android OS zabaleny. V poslední fázi analýzy byly identifikovány informace dostupné z analýzy APK balíčku, které lze použít pro detekci malwaru.

4.1 Datová sada

Datová sada byla složena ze 169 vzorků malwaru, nejstarší vzorek byl přidán v březnu roku 2019 a nejnovější vzorky jsou z ledna roku 2020. Vzorky byly seřazeny dle abecedy ve složkách a následně v nich byly jednotlivé APK balíčky pojmenovány podle výstupní hodnoty SHA256 hašovací funkce, která se vyznačuje tím, že při libovolné délce vstupu vrací pevnou délku výstupu. Na základě této funkce také probíhala kontrola, která byla doplněna o další příznaky sloužící k detekci malwaru. Jednotlivé vzorky pro analýzu byly z datové sady vybírány postupně a to tak, že nejprve byly analyzovány jednodušší a menší aplikace a postupně se složitost analyzovaných aplikací zvyšovala. Datová sada byla poskytnuta společností Monet+, a.s.[2] a je dostupná ke stažení z GitHubu[3]. Je nutno podotknout, že všechny soubory jsou chráněny heslem, které je dostupné ze souboru README.md, pokud by nebyly chráněny heslem, počítačový anti-malware program by je odstranil.

Ve vzorcích bylo použito několik obfuskačních technik, zejména pak namely identifier a string encryption, což výrazně komplikovalo analýzu. Například vzorky v „AndroidMalware_2019-master\Xloader.zip“ se liší zejména výstupní hodnotou hašovací funkce. Tyto změny byly vytvořeny přejmenováním tříd, metod a změnou ikon. Změny zde nemají vůbec žádný vliv na logické fungování aplikace, jedná se tedy o obfuskační techniku namely identifier. Je pravděpodobné, že soubor 466dafa82a4460dcad722d2ad9b8ca332e9a896fc59f06e16ebe981ad3838a6b.apk byl vytvořen jako první, protože v tomto v tomto APK souboru je kód nejčitelnější a nevyužívá tolik obfuskačních technik, jako další dva xLoader APK balíčky 332e68d865009d627343b89a5744843e3fde4ae870193f36b82980363439a425.apk a

[2] Webové stránky společnosti Monet+, a.s.: https://www.monetplus.cz/.
[3] Databáze malwaru: https://github.com/sk3ptre/AndroidMalware_2019.

403401aa71df1830d294b78de0e5e867ee3738568369c48ffafe1b15f3145588.apk. Malware sbírá osobní informace a je využíván k distribuci nakažených aplikací.

Kromě obfuskačních technik byly nalezeny i další obranné mechanismy. Například v souboru „AndroidMalware_2019-master\TVRemote.zip“ byla objevena Anti-Debug technika, která pomocí *Debug.isDebuggerConnected()* kontroluje, jestli je aplikace spouštěna s ladícím nástrojem a kontroluje také jestli je spuštěna ve virtuálním stroji.

4.2 Statická analýza APK

Před samotnou statickou analýzou bylo nejprve potřeba porozumět fungování samotného operačního systému Android, který je specifický v celé řadě aspektů, jako je například:

- mechanismus oprávnění;
- multi-user přístup, ve kterém každá aplikace vystupuje ve svém aplikačním sandboxu jako jeden uživatel pod svým uživatelským ID;
- způsoby podepisování aplikace a další principy.

To samozřejmě pro analýzu nestačí a je třeba znát také fungování jednotlivých komponent aplikací a metod z pohledu vývojáře, potažmo tvůrce mobilního malwaru. Je to důležité zejména proto, že při analýze zdrojového kódu aplikace je nutné rozumět jak jednotlivým metodám, tak aplikaci jako celku.

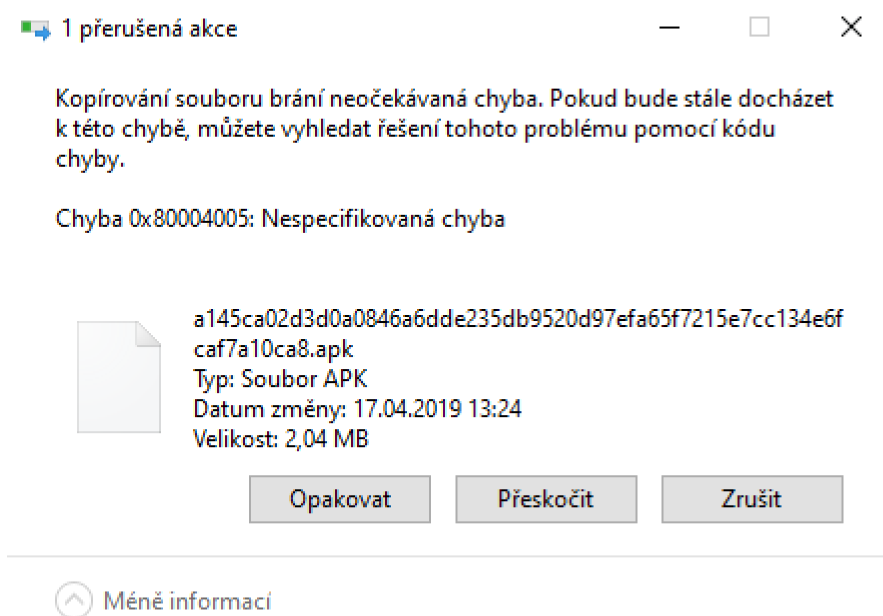
Statická analýza měla za cíl na vzorcích malwaru vypozorovat znaky, které vystihují jednotlivé nakažené aplikace a jsou použitelné pro jeho následnou detekci. Tento proces však není až tak jednoduchý, jak se může na první pohled zdát. Kromě prvního problému, který se týkal nutnosti znát prostředí Android OS se samozřejmě objevovaly další problémy. Některé z nich, jako nutnost dekompilace APK souboru před analýzou je možné vyřešit pomocí speciálních nástrojů. Jiné, jako obfuskační techniky je možné vyřešit velkým množstvím času stráveným nad analýzou potenciálního malwaru.

4.2.1 Nástroje

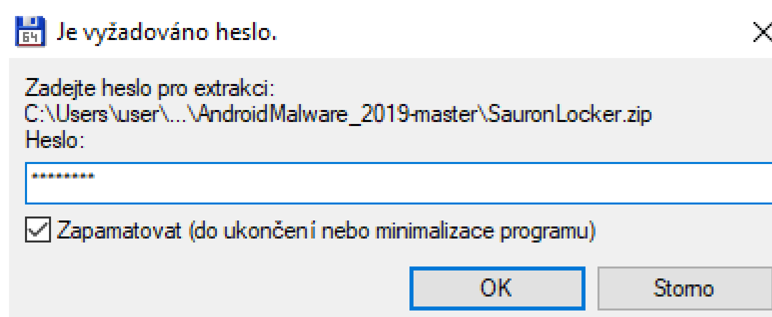
Na základě provedeného výzkumu, byly zkoušeny různé softwarové nástroje. Z nástrojů, které se osvědčily, byla vytvořena analytická sada, jež je popsána níže.

4.2.1.1 Total Commander

Total Commander[4] je na rozdíl od ostatních nástrojů velmi známý. Slouží jako správce souborů, ale jeho využití je velmi široké. Total Commander pomáhal řešit problém se zaheslovaným APK souborem. Pro práci s APK balíčkem je potřeba získat soubory a k tomu je potřeba zadat heslo. Operační systém Windows 10 místo výzvy k zadání hesla vypíše nespecifikovanou chybovou hlášku, což znamená, že není možné soubor dekomprimovat, viz Obr 10.



Obrázek 11 Správce souborů Windows - chybová hláška [zdroj vlastní]



Obrázek 10 Total Commander – žádost o heslo [zdroj vlastní]

Total Commander uživatele automaticky vyzve k zadání hesla k souboru (viz Obr. 11), což znamená, že lze soubor úspěšně rozbalit.

[4] Total Commander (v9.22a): Total Commander. [software]. 29 Mar 2019 [cit. 2020-07-06]. Dostupné z https://www.ghisler.com/ (2019).

4.2.1.2 ApkTool

Nástroj ApkTool[5] slouží k dekompilaci APK balíčku. V praxi to znamená, že obnoví zdroje aplikace do téměř původní podoby pro následnou analýzu. Lze namítnout, že APK balíček je soubor ZIP, takže by mohlo stačit jej dekomprimovat, což je z části pravda, ale prakticky po otevření dekomprimovaného souboru APK nelze přečíst velké množství dat, například soubor AndroidManifest.xml. Tato situace při použití ApkTool nenastane. Použitím příkazu „*apktool d názevAPK*“ se dekóduje APK (Obr. 12 ukazuje použití s parametrem „-o“, který pojmenuje výstupní adresář jako „apktooled“). To má za následek, že AndroiManifest.xml je možné otevřít například v textovém editoru, nebo webovém prohlížeči. ApkTool při dekompilaci rovněž provede baksmaling, čímž se převede dex soubor do smali kódu, poté je opět možné tyto soubory otevřít například pomocí textového editoru. Při manuální analýze byl využíván soubor classes.dex dostupný dekomprimací souboru ZIP. Druhá možnost, jak získat soubor dex, je použití ApkToolu s parametrem „-s“. To má za následek zachování dex souboru v novém adresáři.

```
C:\TestingMalware>apktool d 86507924e47908aded888026991cd03959d1c1b171f32c8cc3ce62c4c45374e
f.apk -o apktooled
I: Using Apktool 2.4.1 on 86507924e47908aded888026991cd03959d1c1b171f32c8cc3ce62c4c45374ef.
apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\user\AppData\Local\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...

C:\TestingMalware>
```

Obrázek 12 Použití nástroje ApkTool [zdroj vlastní]

4.2.1.3 Dex2Jar

Dex2Jar[6] převádí aplikační logiku na JAR soubor. Jsou dvě varianty použití. První variantou je použití nástroje po dekompilaci ApkToolem s parametrem „-s“ Při použití příkazu „*d2j-dex2jar.bat classes.dex*“ na platformě Windows (Obr. 13), jsou konvertovány soubory formátu dex (classes.dex) na formát JAR (ve výchozím stavu vznikne soubor classes-

[5] ApkTool (v2.3.1): ApkTool. [software]. 29 Nov 2019 [cit. 2020-07-01]. Dostupné z https://ibotpeaches.github.io/Apktool/ (2019).

[6] Dex2Jar (v2.0): Dex2Jar. [software]. 10 Nov 2016 [cit. 2020-07-0 1]. Dostupné z https://sourceforge.net/projects/dex2jar/ (2016).

dex2jar.jar). Druhou variantou je příkaz „dex2jar.bat nazevAPK“, což umožní získání JAR souboru přímo z APK balíčku studované aplikace. Tímto se odemyká zdrojový kód aplikace, nicméně kód ještě není přístupný, jelikož formát JAR není možné otevřít nativními nástroji Windows.

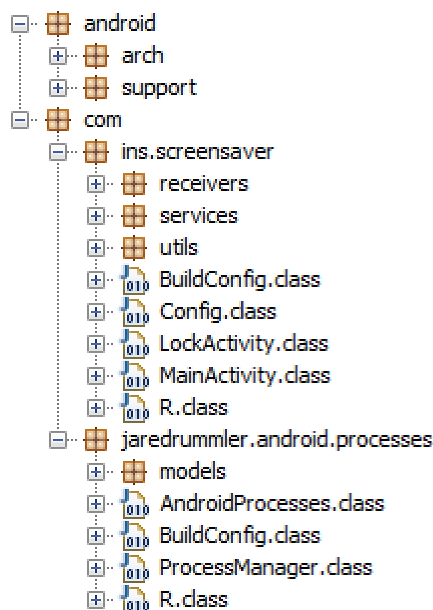
```
C:\Tools\dex2jar-2.0>d2j-dex2jar.bat classes.dex
dex2jar classes.dex -> .\classes-dex2jar.jar

C:\Tools\dex2jar-2.0>
```

Obrázek 13 Použití nástroje Dex2Jar [zdroj vlastní]

4.2.1.4 JD-GUI

Pro vytvoření souboru ve formátu JAR lze využít nástroj Java Decompiler – GUI (JD-GUI[7]). Pro otevření potřebných souborů stačí přes navigační menu otevřít výstupní soubor z nástroje Dex2Jar. Poté jsou načteny balíčky s třídami aplikace, které je možné manuálně analyzovat (Obr. 14). Tento nástroj je vhodný, zejména pro intuitivní a rychlou analýzu tříd a metod aplikace.



Obrázek 14 JD-GUI [zdroj vlastní]

[7] Java Decompiler GUI (v1.6.6): JD-GUI. [software]. 25 Dec 2019 [cit. 2020-07-04]. Dostupné z http://java-decompiler.github.io/ (2019).

4.2.1.5 Mobile Security Framework

Mobile Security Framework (MobSF[8]) nabízí opravdu širokou škálu funkcí pro testování mobilních aplikací. MobSF je rozdělen na dvě hlavní části a sice statickou a dynamickou. I přesto, že práce není zaměřena na dynamickou analýzu, je nutné upozornit na možnost využití nástroje i pro dynamickou analýzu, díky které je možné zkoumat aplikace i za běhu.

MobSF je po stažení možné přímo spustit ze stažené složky pomocí rozkliknutí souboru „run“ a to jak pro Windows, tak i Linux jelikož má připravené, jak dávkové soubory, tak Shell scripty. V příkazové řádce je zobrazena adresa, kterou je možné zadat do webového prohlížeče a zapnout tak MobSF. V adrese je použito číslo portu, které je možné upravit v samotném run souboru. Výchozí hodnota je „8000“. Po zapnutí se aplikace nachází v sekci pro statickou analýzu a je možné ji začít okamžitě používat. Pro analýzu je nutné aplikaci předložit nějaký APK soubor. To se provádí buď vyhledáním jeho MD5 hodnoty, prostřednictvím dialogového okna, nebo pomocí Drag & Drop technologie (tzn., že stačí ze souborového systému přemístit APK do prohlížeče myší). Po skončení analýzy nabídne MobSF komplexní přehled o aplikaci. Framework mimo základních informací, jako název, verze, ikona a hodnoty několika hašovacích funkcí nabízí základě manifestu přehled o komponentách aplikace (také jejich prohlížení a stažení). Framework je rozsáhlý a může poskytnout například kontrolu volaných URL adres vůči blaclistu, nebo API které aplikace volá. Je možné vygenerovat PDF report, který shrnuje vše podstatné a v aplikaci nalezené. Aplikace navíc tvoří historii všech analyzovaných souborů, takže není nutné je analyzovat stále znova. MobSF navíc také často poskytuje vysvětlení, proč je daná informace zajímavá. To velmi pomáhá při procesu učení, často framework najde informace, které se manuálně hledají velmi špatně. Framework pracuje dobře i se základními obfuskačními technikami, což je viditelné na již zmiňovaném malwaru xLoader (viz oddíl 4.1), kde jsou všechny tři reporty totožné v rámci logiky, ale vystupují každá jako úplně jiná aplikace. MobSF také může sloužit k prvotnímu seznámení s datovou sadou, vzhledem k tomu, že ukládá všechny proběhlé analýzy je možné nejprve prozkoumat datovou sadu tímto nástrojem a získat tak hrubý přehled o velikosti a složitosti aplikací.

[8] Mobile Security Framework (v3.0.5 Beta): MobSF. [software]. 13 Mar 2020 [cit. 2020-07-04]. Dostupné z https://github.com/MobSF/Mobile-Security-Framework-MobSF/ (2020).

4.2.2 Ukázka statické analýzy APK – testovací případ SauronLocker

V datové sadě, přesněji v umístění: „AndroidMalware_2019-master\SauronLocker.zip“ se nachází jeden APK balíček. Tento APK balíček nese název ve tvaru výstupní hodnoty hašovací funkce SHA256, což je možné ověřit. APK balíček byl do databáze přidán 17. dubna 2019. Tato aplikace je ransomware, který zablokuje telefon i data a vyžaduje po uživateli platbu pro odemknutí. Před tímto malwarem je samozřejmě možné se chránit. První možností je anti-malware pro detekci takových aplikací. Druhá možnost je pravidelné zálohování, to sice nepředchází nakažení, ale pokud je zařízení nakaženo je možné předejít finanční újmě a ztrátě dat.

1. Odemknutí souboru: Za předpokladu, že datová sada byla již stažena je nutné APK před analýzou nejprve odemknout jelikož je chráněno heslem. Pomocí Total Commanderu je vhodné přesunout APK soubor do „C:\SauronLocker“. Total Commander vyzve k zadání hesla, které je dostupné ze souboru README.

2. Dekompilace APK: Soubor APK je odemknutý, nicméně je ho třeba dekompilovat pomocí nástroje ApkTool. V umístění odemknutého APK se pomocí příkazové řádky zavolá příkaz „*apktool d -s Nazev.apk*“ (Obr. 15).

```
C:\SauronLocker>apktool d -s a145ca02d3d0a0846a6dde235db9520d97efa65f7215e7cc134e6fcaf7a10ca8.apk
I: Using Apktool 2.4.1 on a145ca02d3d0a0846a6dde235db9520d97efa65f7215e7cc134e6fcaf7a10ca8.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\user\AppData\Local\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Copying raw classes.dex file...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...

C:\SauronLocker>
```

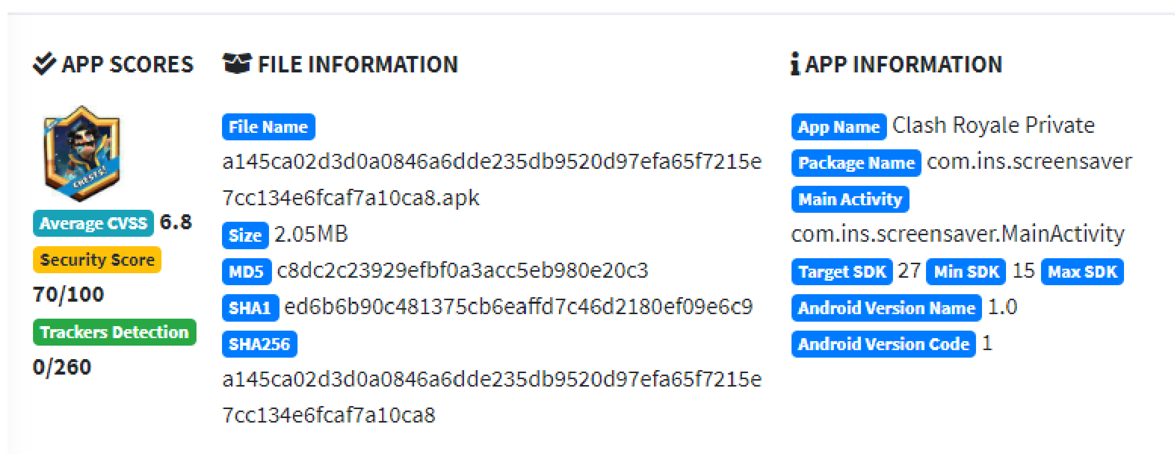
Obrázek 15 Použití ApkTool se zachováním dex souborů [zdroj vlastní]

3. Konverze dex souborů: Vznikla nová složka, která má stejné jméno jako původní APK. Její struktura se od výchozího stavu dekompilované aplikace se liší zejména přítomnosti dex souborů, které normálně ApkTool převádí na smali (Obr. 16). Vytvořený soubor classes.dex je pomocí Dex2Jar konvertován na soubor classes-dex2jar.jar. To bylo provedeno zadáním příkazu „*d2j-dex2jar.bat classes.dex*“ do příkazové řádky (předpokládá se, že soubor classes.dex byl přemístěn do složky ve které se nachází Dex2Jar).

original	06.07.2020 18:13	Složka souborů	
res	06.07.2020 18:13	Složka souborů	
AndroidManifest	06.07.2020 18:13	Soubor XML	2 kB
apktool.yml	06.07.2020 18:13	Soubor YML	2 kB
classes.dex	06.07.2020 18:13	Soubor DEX	3 003 kB

Obrázek 16 Nová složka s dex souborem [zdroj vlastní]

4. MobSF analýza: Pomocí souboru run a přesměrování na adresu localhost s příslušným portem lze spustit MobSF. Následným přetažením apk souboru je možné provést analýzu. Ihned po dokončení jde vidět několik zajímavých informací, například je zde možné si ověřit, že soubor je pojmenován opravdu dle SHA256 (Obr. 17).



Obrázek 17 MobSF – Základní informace [zdroj vlastní]

První čtyři kroky lze aplikovat obecně na všechny APK soubory, následující kroky se liší na základě komponent, které aplikace obsahuje a funkcionalit, které poskytuje. Využívání MobSF je ideální zároveň s manuální analýzou, kdy umožňuje kontrolovat získané informace. V této fázi může začít analýza, jelikož APK soubor, potažmo vzniklá složka, je již v potřebném formátu pro zkoumání.

5. Analýza manifestu: Na obrázku 18 je vidět, že soubor AndroidManifest.xml poskytuje celou řadu užitečných informací. Jeho otevření je možné provést ze složky „C:\SauronLocker“ (viz 1. Odemknutí souboru) pomocí webového prohlížeče, nebo textového editoru. Nejprve je uveden název balíčku následovaný řadou oprávnění. Ta jsou u této aplikace velice ojedinělé a žádají o přístup jak k externímu úložišti, tak i ke kontaktům. Důvěryhodně nepůsobí ani oprávnění pro získání informace o dokončení bootování, které často využívají ransomwary pro uzamknutí telefonu ihned po zapnutí.

```
▼<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.ins.screensaver">
   <uses-permission android:name="android.permission.INTERNET"/>
   <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
   <uses-permission android:name="android.permission.SET_WALLPAPER"/>
   <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
   <uses-permission android:name="android.permission.READ_CONTACTS"/>
   <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
 ▼<application android:allowBackup="true" android:debuggable="true"
   android:icon="@drawable/icon" android:label="Clash Royale Private"
   android:largeHeap="true" android:roundIcon="@drawable/icon"
   android:theme="@style/Theme.AppCompat.Light.NoActionBar">
   ▼<activity android:name="com.ins.screensaver.MainActivity">
     ▼<intent-filter>
         <action android:name="android.intent.action.MAIN"/>
         <category android:name="android.intent.category.DEFAULT"/>
         <category android:name="android.intent.category.LAUNCHER"/>
       </intent-filter>
     </activity>
     <activity android:name="com.ins.screensaver.LockActivity"/>
   ▼<receiver android:name="com.ins.screensaver.receivers.OnBoot"
     android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
     ▼<intent-filter>
         <action android:name="android.intent.action.BOOT_COMPLETED"/>
         <action android:name="android.intent.action.QUICKBOOT_POWERON"/>
       </intent-filter>
     </receiver>
     <service android:name="com.ins.screensaver.services.CheckerService"/>
   </application>
 </manifest>
```

Obrázek 18 AndroidManifest.xml [zdroj vlastní]

Další zajímavou položkou je název aplikace – „Clash Royal Private“, jedná se o velmi populární hru a slovo Private naznačuje možnost využití placených funkcí zdarma. Doplněním Mobile Security Frameworku je možné se podívat na ikonu, která naznačuje použití techniky repackaging. Ve skutečnosti tomu tak není a je zde pouze zneužit název a ikona legitimní aplikace.

Z manifestu je také patrné, že aplikace se skládá ze dvou aktivit, jednoho receiveru a jedné služby. V této aplikaci je již z manifestu zřejmé, co se bude dít. Receiver deklarovaný v manifestu jasně naznačuje, že slouží k informování aplikace ihned potom, co zařízení nabootuje. Typické chování uživatele při tomto malwaru je totiž restartování telefonu, to však díky tomuto receiveru nepomůže.

6. Analýza JAR: V dalším kroku byl vytvořen soubor classes-dex2jar.jar a nyní je nutné ho otevřít nástrojem JD-GUI. Zobrazí se třídy, se kterými aplikace pracuje. V tomto kroku lze

využít i MobSF, ale JD-GUI nabízí klasickou strukturu známou z vývojových prostředí a tak v něm může být pro vývojáře jednodušší pracovat.

Aktivity jsou zde dvě – první je MainActivity.class, která již v manifestu měla příznak LAUNCHER, což znamená, že se spouští jako první. Tato aktivita slouží zejména pro spuštění druhé aktivity a sice LockActivity.class – uvedená třída je velmi obsáhlá a jsou v ní hledané kódy provádějící škodlivou činnost. Ideálním způsobem, jak s takovou třídou naložit je automatizované zaznamenání sekvence API volání. Avšak i manuálně je možné zjistit celou řadu informací. Nejdůležitějšími z nich je třída zobrazující zprávu, ve které vyzývá uživatele k zaplacení určité sumy peněz a implementace metody pro zašifrování a dešifrování všech adresářů, souborů a kontaktů.

Aplikace má ještě další dvě důležité komponenty – receiver a službu (service). Služba zde kontroluje, jestli je aplikace na popředí a pokud ne, učiní tak. U receiveru se potvrzuje teorie, že receiver zde slouží k zapnutí LockActivity.class ihned po dokončení startu zařízení.

To však samozřejmě stále není vše, při bližší analýze kódu se vyskytuje velké množství volání několika metod, které nejsou standardně dostupné z Java API Frameworku. Jedná se o funkcionalitu zapouzdřenou do knihovny napsané tvůrcem malwaru. Velmi často se jedná o kvalitní kódy, u kterých je škoda, že jsou takto zneužívány. Ve zkoumané aplikaci je několik podpůrných tříd a vzhledem k tomu, že kód není nikterak obfuskován je i čitelný. Aplikace má algoritmus pro šifrování a dešifrování, který se nachází ve třídě AES.java (což je zkompilovaný soubor .class). Další klíčová funkcionalita se nachází ve třídách končících slovem Util, například FileUtil.java slouží pro načtení a přepsání souborů a analogicky je to samozřejmě i s dalšími třídami, jako ContactUtil.java. Další důležitou částí je balíček *„com\jaredrummler\android\processes\“* ten slouží hlavně pro získání všech aktuálně běžících aplikací. Zajímavostí je, že tvůrce mobilního malwaru využívá knihovnu autora Jared Rummler. Autor této knihovny ale vyvinul i APK parser, který v detekční aplikaci pomáhá získat data k odhalení malwaru.

7. Analýza ostatních souborů a vlastností: V testované aplikaci jsou důležité stringy, které se nacházejí v „nazevAPK\res\values\strings.xml“. V MobSF je možné vidět, že několik metod tyto stringy používá pro komunikaci s doménami. MobSF tyto domény neoznačil za škodlivé, ale vzhledem k tomu, že se objevují v malwaru je vhodné je zahrnout do analýzy.

Ve vyšetřovaném APK souboru se kromě stringů v „nazevAPK\res\values\strings“ nevyskytují další potenciální příznaky.

MobSF také upozorňuje, že našel Anti VM code – to je detekce emulátoru. Tvůrci malwaru často používají kontrolu, zda je jejich aplikace analyzována/spouštěna v emulátoru.

V některých malwarech je také objevují binární soubory v adresáři assets, které slouží k dynamickému načítání komponent aplikace za běhu. Ty pak lze zahrnout mezi potenciální příznaky.

4.3 Navrhnutí příznaků použitelných pro klasifikaci malwaru

Příznaky, které lze použít pro klasifikaci malwaru není snadné vybrat. Například oprávnění v SauronLocker by se jistě dala využít pro klasifikaci malwaru. Problém nastává, pokud má aplikace méně oprávnění. Pokud by měla aplikace například jen oprávnění pro používání internetu, pak by mohly být všechny aplikace, které využívají jen oprávnění k internetu chybně vyhodnoceny jako malware, tedy klasifikace by vyvolala falešný poplach.

Ze studia prací, které se věnují této problematice vyplynula skutečnost, že je ideální kombinace několika metod. Například pokud by aplikace měla pouze oprávnění pro přístup na internet, ale jako doplněk ke klasifikaci na základě oprávnění by bylo použito testování na základě volání API, nebo certifikátu je daleko menší šance, že bude vyvolán falešný poplach. Obecně lze říci, že je výhodné při analýze sbírat co nejvíce informací o vyšetřovaných aplikacích. Mobilní malware má různorodé znaky, proto je důležité si vytvořit o APK co největší databázi informací, třebaže nejsou hned využity pro klasifikaci, je možné že časem budou pro nějakou rodinu malwaru potřebné.

A. Výstupní hodnota hašovací funkce a Package name: Klasifikace na základě SHA256 může být vyhodnocena pravdivostní hodnotou, tedy pravda a nepravda. Pokud byl již malware detekován někdy dříve poskytuje stoprocentní jistotu, že bude odhalen, za předpokladu, že nebyl malware modifikován. SHA256 je velmi spolehlivá ochrana, byť má i svá omezení, například stačí malá změna ve zdrojovém kódu, jako název proměnné a hodnota SHA256 nově zkompilovaného APK se změní.

Podobné je to u package name, tedy jména balíčku. Zde opět funguje podobný princip, pokud byl malware již detekován a jeho název nebyl změněn, pak je možné na základě booleanovské logiky říct, zda se jedná o malware, nebo ne. Největším rozdílem a problémem oproti klasifikací dle SHA256 je možnost vyvolání falešného poplachu a to pokud by se název malwaru shodoval s názvem legitimní aplikace. Sice by došlo ke správné klasifikaci malwaru, ale zároveň k nesprávné klasifikaci neškodlivé aplikace.

Tyto dvě metody klasifikace jsou ideální pro rychlé ověření na základě textových řetězců, které mají jako svůj výstup logickou hodnotu. Proto se při klasifikaci hodí zejména jako první krok, pokud by už APK bylo detekováno, je zbytečné vyhodnocovat ostatní příznaky. Neplatí to však naopak, což znamená, že pokud ani jedna z těchto kontrol nevyhodnotí aplikaci jako malware je nutné dále kontrolovat její příznaky.

B. Základní informace: do této kategorie lze zařadit i již zmíněný název balíčku. Dalšími informacemi v této kategorii může být ikona, verze SDK, název aplikace. Uvedené informace jsou vhodné hlavně pokud jsou tvořeny reporty, je následně možné porovnat třeba dvě různé verze stejného malwaru. Nicméně jen na základě těchto dat není možné provádět spolehlivou klasifikaci a to ze stejných důvodů, které byly popsány u package name.

C. Oprávnění: Aby aplikace mohla přistupovat k určitým částem frameworku, potřebuje zažádat a obdržet oprávnění. Na základě oprávnění, které aplikace vyžaduje lze předpokládat, jaké metody bude využívat, potažmo jaké funkcionality bude poskytovat. Sada zmíněná v příkladu malwaru SauronLocker je velmi specifická a neobvyklá, což znamená, že by celou sadu bylo možné použít jako příznak pro klasifikaci.

Sadu oprávnění je možné pro zrychlení klasifikace uložit jako výstupní hodnotu hašovací funkce, nicméně to bude mít za následek selhání při změně pořadí oprávnění, což je nevýhodné. Uvedený problém lze vyřešit setříděním oprávnění, nicméně nebude možné určit míru shody (viz Příklad 1), proto je lepší oprávnění nehašovat.

Příklad: Nechť je aplikace A (posuzovaná) mající šest oprávnění (shodná s B) a aplikace B (vzor malware) mající sedm oprávnění (všechna z B a jedno navíc). V tomto stavu při využití pattern matchingu je shoda vyhodnocena jako nepravda. Může však jít o uměle přidané oprávnění sloužící ke skrytí malwaru.

Za cenu výkonu je tedy lepší s oprávněními nakládat jako s textovými řetězci, které mohou měnit pořadí. S takovými řetězci lze naložit dvěma způsoby – první je zjištění oprávnění určitého počtu nenakažených a nakažených aplikací a následně způsobem, který je popsán v teoretické části vyhodnotit kolik oprávnění bylo označeno jako „nakažené oprávnění“ (tedy oprávnění, které se objevuje v nakažených aplikacích). Druhá možnost je pracovat pouze s aplikacemi nakaženými a sbírat oprávnění jako celé sady. Tato možnost dobře funguje na malwarech, jako je SauronLocker, ale může vyvolávat falešné poplachy při nižším počtu oprávnění, proto je ideální ji zkombinovat s některou další metodou, například analýzou dat z tagů uses-feature.

D. Uses-feature: Na základě tohoto tagu nelze postavit kompletní klasifikaci především kvůli tomu, že velké množství aplikací včetně malwarů tento tag vůbec nepoužívá. Ovšem jedná se o hodnotný doplněk k analýze oprávněním. Například v situaci, kdy malware má stejná oprávnění jako legitimní aplikace, ale malware nemá žádné uses-feature tagy (na rozdíl od čisté aplikace). V takových případech může analýza Uses-feature tagů to pomoci ke snížení počtu falešných poplachů.

E. Komponenty aplikace: Z hlediska klasifikace lze na komponenty pohlížet buď tak, že jsou to pouze textové řetězce. Na základě textových řetězců je kontrolováno, zda se v kontrolované aplikaci nacházejí nějaké škodlivé komponenty, případně jestli se komponenty shodují. Tato metoda sice funguje relativně dobře, ale nekontroluje obsah komponent, protože tyto informace jsou získávány z Manifestu. Tato metoda selže v případě, že budou třídy obfuskovány, nebo když by malware napodobil strukturu a názvy komponent legitimních aplikací. To je nežádoucí, proto je nutné uvedenou metodu opět zkombinovat. V tomto případě se jako vhodný doplněk jeví kontrola sekvence volání API, jelikož ověří i kód jednotlivých komponent.

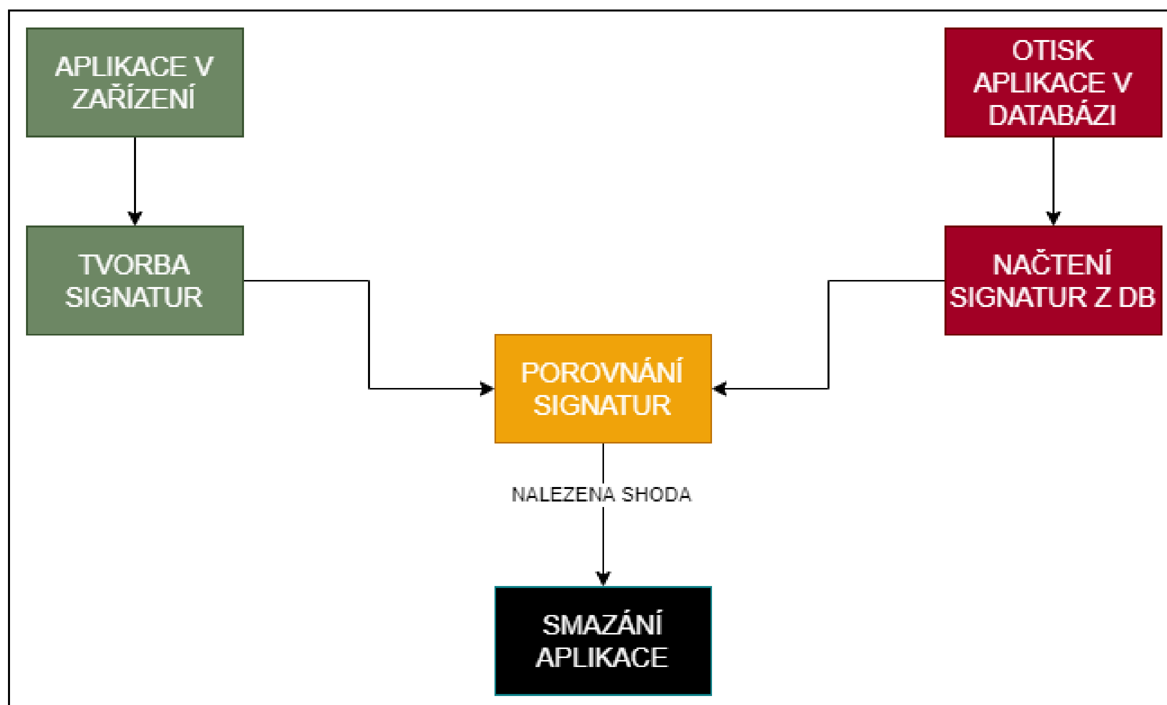
F. Sekvence volání API: Tato metoda je jedna z nejefektivnějších. Vyrovná se i s obfuskováním, jelikož zaznamenává, jaké metody se volají. Když je tato klasifikace implementována, často bývají ukládány sekvence volaných metod dle ID namísto názvu. Nevýhodou uvedené metody je hlavně implementace, jelikož je nutné vytvořit parser, který zaznamenává v jakých komponentách bylo provedeno volání (a to včetně názvu volané metody). To sice dobře vystihuje, jak funguje celá aplikace, ale je to výpočetně náročné, jelikož se musí analyzovat velké množství kódu. V současné době již existují techniky umožňující vytvářet malware rezistentní na tento typ testů. Jelikož je zaznamenávána sekvence volání, tedy jak jsou jednotlivé metody po sobě volány je možné vložit například kód, který nemá vliv na logiku aplikace, ale změní sekvenci volání.

G. Informace o tvůrci: Všechny dosud představené způsoby klasifikace hodnotily pouze samotný malware a nepřihlížely na to, kdo jej napsal. Vzhledem k tomu, že před distribucí je nutné aplikaci podepsat vývojářem je možné tohoto faktu využít. Pomocí parserů je možné získat informace o certifikátu. Potřebná informace o certifikátu je jeho „Serial number“. Často se totiž tato sériová čísla certifikátů u malwarů opakují, důkazem může být například vzorek SlockerWannacry z datové sady. Zde ze tří APK balíčků (200d8f98c326fc65f3a11dc5ff1951051c12991cc0996273eeb9b71b27bc294d.apk; 2ffd539d462847bebcdff658a83f74ca7f039946bbc6c6247be2fc62dc0e4060.apk;

36f40d5a11d886a2280c57859cd5f22de2d78c87dcdb52ea601089745eeee494.apk) mají dva stejný certifikát.

5 DETEKCE MALWARU

Detekce malwaru probíhá pomocí implementované aplikace na mobilním zařízení. Aplikace využívá signature-based přístupu a umí sbírat data o aplikacích – tedy signatury a provést následnou detekci. Díky tomu je možné analyzovat aplikace nainstalované v telefonu, a poté je porovnat vůči vzorům z předem vytvořené databáze, která je součástí detekční aplikace (Obr. 19).



Obrázek 19 Schéma detekční části [zdroj vlastní]

5.1 Nástroje

Android Studio[9]: Vývojové prostředí nabízí všechny potřebné funkcionality pro potřeby vývoje detekční aplikace. Prostředí poskytuje i virtuální zařízení - emulátor, ten je možné využít k testování vyvíjených aplikací. Navíc IDE obsahuje i tzv.: Device File Explorer, který slouží k práci se soubory v telefonu a poskytuje grafické prostředí pro souborový systém pro Android OS. Díky tomu je možné například sledovat nainstalované aplikace v zařízení, nebo kontrolovat databáze.

[9]Android Studio SW dostupný z: https://developer.android.com/studio/?gclid=Cj0KCQjw6575BRCQARIsAMp-ksMThM4DyHbwGXB7XR5lRPg-w2ed1wG8rDpbWaIXlDB4dfbc-1HfyK4aAmNaEALw_wcB&gclsrc=aw.ds.

SqLiteStudio[10]: Jedné se o nástroj sloužící k zobrazení databáze, podporuje používaný formát SqLite verzi 3. To znamená, že zobrazí v přehledném grafickém prostředí data a struktury databáze.

GitHub desktop[11]: Tento nástroj slouží k verzování aplikace. Umožňuje ukládat software na cloudové úložiště s poznámkami, jaké změny byly provedeny. Díky tomu je program zálohovaný a k daným verzím aplikace je možné v případě potřeby libovolně přistupovat.

5.2 Implementace

Detekční aplikace se skládá ze tří hlavních modulů (Obr. 19 – Modul A: zelená; Modul B: červená; Modul C: oranžová, černá). Modul A slouží k vytvoření databáze signatur, případně její aktualizaci. Modul B tyto signatury čte z databáze a vytváří instance třídy SignatureSetModel, se kterými v další fázi pracuje modul C při detekci. Program byl napsán v programovacím jazyce Java s využitím Java API frameworku pro operační systém Android. Aplikace využívá externí knihovny získané z GitHubu, které poskytují nezbytné funkcionality, nebo usnadňují práci například APKParser[12] slouží k práci s APK soubory. Kromě APKParser knihovny byl využit i modul pro konverzi datového typu JSON – GSON[13].

Aplikační logika je umístěna ve třech hlavních modulech. Grafické prostředí se skládá ze dvou obrazovek, přičemž jedna slouží jako úvodní displej – tzv. splashscreen a druhá aktivita obsluhuje interakci uživatele. V souboru AndroidManifest.xml jsou deklarovány obě aktivity, přičemž splashscreen - SplashActivity (Obr. 20) je deklarována s intent-filterem, který má za následek spuštění této aktivity, jako první – vytvoří tzv. entry point (vstupní bod aplikace). Dále je prostřednictvím manifestu požádáno o oprávnění ke smazání aplikací ("android.permission.REQUEST_DELETE_PACKAGES").

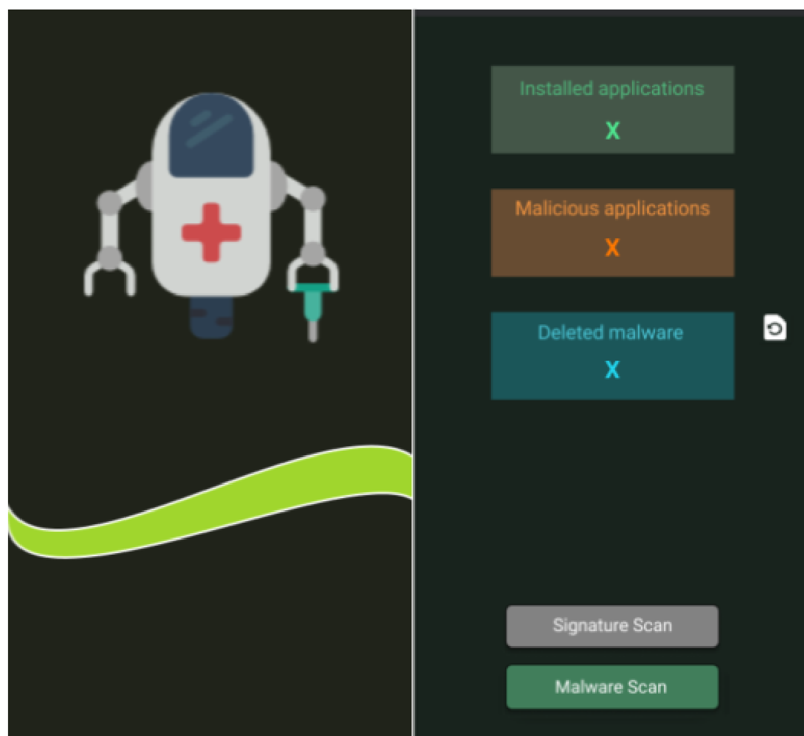
```
<activity android:theme="@style/SplashTheme"
    android:name=".SplashActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
```

Obrázek 20 Deklarace aktivit [zdroj vlastní]

[10] SqLiteStudio SW dostupný z: https://sqlitestudio.pl/.
[11] GitHub Desktop SW dostupný z: https://desktop.github.com/.
[12] Knihovna dostupná z: https://github.com/jaredrummler/APKParser.
[13] Knihovna dostupná z: https://github.com/google/gson.

5.2.1 Grafické uživatelské rozhraní

Splashscreen aktivita slouží zejména jako designový prvek, jehož cíl je upoutat pozornost uživatele při načítání aplikace (Obr. 21 vlevo). Všechny části GUI byly navrženy pomocí webového grafického editoru Figma[14].



Obrázek 21 GUI [zdroj vlastní]

Druhá obrazovka je složena ze tří tlačítek a tří informačních sekcí (Obr. 21 vpravo). Informační sekce jsou obrázky, ve kterých je vložen text (Obr. 21 – označeno X). První sekce informuje uživatele o počtu nainstalovaných uživatelských aplikací – tento počet je zobrazen hned po spuštění aplikace. Druhá část informuje o počtu infikovaných programů v zařízení a je aktualizován po stisknutí tlačítka Malware Scan – ve výchozím stavu je vypsán řetězec, který vyzve uživatele ke stisknutí tlačítka. Poslední sekce vypisuje počet úspěšně smazaných malwarů. Pro aktualizaci hodnoty je vyžadováno stisknutí re-fresh tlačítka. Tlačítko je zde nutné, jelikož se jedná o prototyp u kterého je nutné provádět test stavu aplikací na zařízení, v produkční aplikaci by byl kód umístěn do metody *onResume()*, nebo by byla využita metoda *startActivityForResult()*. Tlačítko ověří, jestli aplikace, které měly být odinstalovány jsou, nebo nejsou nainstalovány. Počet odinstalovaných aplikací je vypsán do příslušného textového pole. Poslední částí GUI je tlačítko Signature Scan, které

[14] Figma SW dostupný z: https://www.figma.com/.

je ve výchozím stavu zakázáno a slouží pro vývojářské účely (pro individuální přidání vzorků do databáze).

5.2.2 Modul A – Tvorba signatur z aplikace na zařízení

Signatury aplikací získává modul A aplikační logikou umístěnou ve třídě MainActivity.java, přesněji v metodě *onCreate(),* kde je implementována metoda *setOnClickListener()* pro tlačítko Signature Scan.

```
btn_signatureScan.setOnClickListener((view) → {

        //SCAN ALL PHONE APPS
        PackageManager pm = getPackageManager();
        List<PackageInfo> InstalledApps = GetNonSystemApks();
        boolean success;
        //CYCLE THROUGH EACH APK
        for(int apkNumber = 0; apkNumber < InstalledApps.size(); apkNumber++){

            //ACTIVE APP TO SCAN
            PackageInfo loadedApp = InstalledApps.get(apkNumber);
            packageName = loadedApp.packageName;
            //IGNORE MY APPS
            if(packageName.toLowerCase().equals("cz.bachelor.sqlite")
                    ||packageName.equals("cz.bachelor.detector")){
                continue;
            }

            //EACH APP HAS SIGNATURE SET MODEL WHICH SPECIFICATES IT
            SignatureSetModel ss = getSignatureSetModel(pm, apkNumber, packageName);
```

Obrázek 22 Obsluha tlačítka [zdroj vlastní]

Proces vytváření signatur je zahájen stisknutím tlačítka Signature Scan. Je nutné vytvořit instanci třídy Package Manager pro práci s balíčky v Android OS, následně je vytvořen seznam všech nainstalovaných uživatelských aplikací. Uvedený seznam je vytvořen pomocí metody *GetNonSystemApks()*. Metoda vytvoří složený datový typ List (seznam) obsahující informace o nainstalovaných nesystémových aplikacích – nejdůležitější informaci představuje package name. Tento seznam je procházen v cyklu, který ignoruje detekční program vytvořený pro účely bakalářské práce. Tímto se zamezí vnesení chyby do databáze v další fázi.

5.2.2.1 SignatureSetModel

Pokud se podaří najít nainstalovanou uživatelskou aplikaci, je vytvořena instance třídy SignatureSetModel. Instance třídy SignatureSetModel je naplněna daty, které popisují danou aplikaci. Tyto informace jsou získávány pomocí metody *getSignatureSetmodel()*, která jako

parametry přijímá: instanci třídy PackageManager, Integer s číslem aplikace a package name (slouží pro identifikaci APK souboru se kterým pracuje). Metoda *getSignatureSetmodel()* pracuje s instancí třídy APKParser. Díky APKParseru je možné získávat informace o jednotlivých částech APK balíčku na základě jeho package name. Metoda *getSignatureSetmodel()* pracuje na třech částech APK – certifikát, meta a manifest (Obr. 23).

```
//CERTIFICATE CALLS
CertificateMeta cm = null;
try {
    if (apkParser.verifyApk() == ApkParser.ApkSignStatus.SIGNED){
        certificateMD5 = apkParser.getCertificateMeta().certMd5;
    }
} catch (IOException | CertificateException e) {
    e.printStackTrace();
}

//META CALLS
List<UseFeature> useFeatures = meta.usesFeatures;
List<String> requestedPermissions = meta.usesPermissions;

//MANIFEST CALLS
List<AndroidComponent> acts = androidManifest.activities;
ArrayList<String> actNames =  new ArrayList<>();
List<AndroidComponent> prov = androidManifest.providers;
ArrayList<String> provNames =  new ArrayList<>();
List<AndroidComponent> serv = androidManifest.services;
ArrayList<String> servNames =  new ArrayList<>();
List<AndroidComponent> recs = androidManifest.receivers;
ArrayList<String> recNames =  new ArrayList<>();
ArrayList<String> usesFtrNames = new ArrayList<>();
```

Obrázek 23 Práce na částech APK [zdroj vlastní]

V hlavní fázi metoda pomocí cyklů vytvoří seznamy signatur pro snadnější manipulaci s daty. Z těchto seznamů jsou vytvořeny řetězce (kvůli zápisu do DB). Všechny informace jsou uloženy do instance třídy SignatureSetModel, která slouží k popisu jednotlivých aplikací. SignatureSetModel obsahuje konstruktor, gettery, settery a přepsanou metodu *toString()* pro výpis. Dále jsou zde implementovány metody pro konverzi seznamu na řetězec a zpět. Třída SignatureSetModel ukládá následující data: MD5 hodnotu, package name, permissions, use-features, activity, services, receivers a providers.

Ukládání složených datových typů do SQLite databází je problematické, nicméně tento datový typ umožňuje dobrou manipulaci se signaturami a proto je vhodné jej zachovat. Problém s složenými datovými typy je vyřešen pomocí knihovny GSON. Tato knihovna umožňuje vytvořit ze seznamu řetězců jeden řetězec ve formátu JSON. Zpětně knihovna čte řetězec ve formátu JSON a vytváří původní seznam řetězců (Obr. 24). Metody pro konverzi

se jmenují: *StringsToLists()* a *ListsToStrings()*. Obě mají výsledek pravda-nepravda kvůli ověření při vývoji a jsou ošetřeny pomocí try-catch.

```
boolean StringsToLists(String permString, String featureString, String activityString,
                    String serviceString, String providersString, String receiversString){

    boolean result;
    try {
        Type type = new TypeToken<ArrayList<String>>() {}.getType();
        Gson gson = new Gson();
        requestedPermission = gson.fromJson(permString, type);
        usesFeatureName =  gson.fromJson(featureString, type);
        activityNames = gson.fromJson(activityString, type);
        servicesNames = gson.fromJson(serviceString, type);
        providersNames = gson.fromJson(providersString,type);
        receiversNames = gson.fromJson(receiversString,type);
        result = true;
    } catch (JsonSyntaxException e) {
        e.printStackTrace();
        return false;
    }
    return result;
}
```

Obrázek 24 Konverze pomocí GSON [zdroj vlastní]

5.2.2.2 Databáze

S vytvářením signatur souvisí také založení databáze. Díky implementaci v modulu A je možné vytvořit databázi signatur. Na konci aplikační logiky tlačítka Signature Scan je umístěn kód, který přidává signatury právě zpracovávané aplikace do databáze (Obr. 25). Všechny metody, které pracují s databází vytváří SQLite příkazy uložené jako datový typ String.

```
//EACH APP HAS SIGNATURE SET MODEL WHICH SPECIFICATES IT
SignatureSetModel ss = getSignatureSetModel(pm, apkNumber, packageName);
//WRITE TO DB
DataBaseHelper dataBaseHelper = new DataBaseHelper( context: MainActivity.this);
success = dataBaseHelper.addOne(ss);
```

Obrázek 25 Přidání signatury do DB [zdroj vlastní]

Z obrázku 25 je patrné, že k obsluze databáze se využívá třída DataBaseHelper, která implementuje metody pro vytvoření databáze, načtení databáze a přidání záznamu do databáze. Třída je potomkem třídy SQLiteOpenHelper, která se využívá k práci s databázemi SQLite. Práci s databází je možné rozdělit na následující tři části:

1. Vytvoření databáze: tvorba databáze probíhá implementací metody *onCreate()* ve třídě DataBaseHelper. Metoda nemá žádnou návratovou hodnotu a její logika spočívá ve vytvoření textového řetězce tak, aby tvořil SQL příkaz pro vytvoření databáze. Tento příkaz se vykoná na databázi, která byla předána jako parametr funkce (Obr. 26).

```
@Override
public void onCreate(SQLiteDatabase db) {

    String createTableStatement = "CREATE TABLE " + MALWARE_TABLE +
            " (" + COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
            + COLUMN_MALWARE_MD5 + " TEXT, "
            + COLUMN_MALWARE_PACKAGENAME + " TEXT, "
            + COLUMN_MALWARE_PERMISSIONS + " TEXT, "
            + COLUMN_MALWARE_FEATURES + " TEXT, "
            + COLUMN_MALWARE_ACTIVITIES + " TEXT, "
            + COLUMN_MALWARE_SERVICES + " TEXT, "
            + COLUMN_MALWARE_PROVIDERS + " TEXT, "
            + COLUMN_MALWARE_RECEIVER + " TEXT, UNIQUE (MALWARE_MD5) ON CONFLICT REPLACE)";

    db.execSQL(createTableStatement);
```

Obrázek 26 Vytvoření databáze [zdroj vlastní]

2. Přidání záznamu do databáze: *addOne()*, tato metoda slouží pro zápis do databáze, jejím parametrem je instance třídy SignatureSetModel (informace o APK). *addOne()* má návratovou hodnotu boolean pro ověření zápisu do databáze. Metoda využívá třídu ContentValue, která umožňuje s daty pracovat v párových hodnotách – (jméno, hodnota). Tyto dvojice jsou předány spolu s názvem databáze metodě *insert()*. Do proměnné insert, je uložen výsledek operace *insert()*. Je-li hodnota proměnné insert rovna -1, pak metoda *addOne()* vrací false, jinak je návratovou hodnotou true (Obr. 28).

```
boolean addOne(SignatureSetModel ssModel){

    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues cv = new ContentValues();

    cv.put(COLUMN_MALWARE_MD5, ssModel.getCertificateMD5());
    cv.put(COLUMN_MALWARE_PACKAGENAME, ssModel.getPackageName());
    cv.put(COLUMN_MALWARE_PERMISSIONS, ssModel.getRequestedPermissionsString());
    cv.put(COLUMN_MALWARE_FEATURES, ssModel.getFeaturesString());
    cv.put(COLUMN_MALWARE_ACTIVITIES, ssModel.getActivitiesString());
    cv.put(COLUMN_MALWARE_SERVICES, ssModel.getServicesString());
    cv.put(COLUMN_MALWARE_PROVIDERS, ssModel.getProvidersString());
    cv.put(COLUMN_MALWARE_RECEIVER, ssModel.getReceiversString());

    long insert = db.insert(MALWARE_TABLE, nullColumnHack: null, cv);
    if(insert == -1){
        return false;
    }else{
        return true;
    }

}
```

Obrázek 28 Metoda *addOne()* [zdroj vlastní]

3. Čtení z databáze: *getEverything()*, jak název napovídá tato metoda vrátí všechny záznamy z databáze, výsledek je List<SignatureSetModel>. Zajímavé je také využití třídy Cursor, která slouží jako ukazatel v tabulce a je možné díky němu pracovat s jednotlivými sloupci v tabulce (Obr. 27). To znamená, že dle indexů je možné pracovat s každým sloupcem zvlášť. Ještě před vrácením seznamu je nutné provést konverzi některých proměnných, jelikož SQLite neumožňuje ukládat například List<String> je nutné provádět tuto konverzi popsanou v minulém oddílu (viz. oddíl 5.2.2.1).

```
String md5 = cursor.getString( columnIndex: 1);
String packageName = cursor.getString( columnIndex: 2);
String permissionsString = cursor.getString( columnIndex: 3);
String featuresString = cursor.getString( columnIndex: 4);
String activitiesString = cursor.getString( columnIndex: 5);
String servicesString = cursor.getString( columnIndex: 6);
String providersString = cursor.getString( columnIndex: 7);
String receiversString = cursor.getString( columnIndex: 8);
```

Obrázek 27 Přístup ke sloupcům DB [zdroj vlastní]

5.2.3 Modul B – Načtení signatur z otisků aplikací v databázi

Modul B pracuje po stisknutí tlačítka Malware Scan, to znamená že aplikační logika modulu je umístěna v metodě *setOnClickListener()*.

V první fázi jsou načteny nesystémové aplikace a je vytvořena instance třídy DataBaseHelper. Pomocí metody *getEverything()* volané na instanci třídy DataBaseHelper je naplněn List<SignatureSetModel>, tím je načtena databáze signatur (Obr. 29).

```
List<String> malicious = new ArrayList<>();
List<String> deleted = new ArrayList<>();
PackageManager pm = getPackageManager();
List<PackageInfo> InstalledApps = GetNonSystemApks();
int installedApps = InstalledApps.size()-1;

DataBaseHelper dataBaseHelper = new DataBaseHelper( context: MainActivity.this);
List<SignatureSetModel> dbPatterns = dataBaseHelper.getEverything();
```

Obrázek 29 Načtení seznamu signatur [zdroj vlastní]

Modul B pokračuje podmínkou pro ověření počtu uživatelských aplikací (bez detekčního programu). Pokud nejsou žádné aplikace nalezeny zobrazí se notifikace – tzv. toast a metoda se ukončí. Jestliže na zařízení existují i jiné uživatelské programy, než detekční software provede se for cyklus, který prochází všechny nainstalované aplikace (Obr. 30).

```
for(int apkNumber = 0; apkNumber<InstalledApps.size();apkNumber++ ){

    SignatureSetModel deviceApp = new SignatureSetModel();
    SignatureSetModel dbApp = new SignatureSetModel();

    PackageInfo loadedApp = InstalledApps.get(apkNumber);
    packageName = loadedApp.packageName; //nemazat
    if(packageName.toLowerCase().equals("cz.bachelor.sqlite")
            ||packageName.equals("cz.bachelor.detector")){
        continue;
    }
    deviceApp = getSignatureSetModel(pm,apkNumber, packageName);

    for(int dbAppCount = 0; dbAppCount < dbPatterns.size(); dbAppCount++){


        dbApp = dbPatterns.get(dbAppCount);

        isMalware(deviceApp,dbApp,malicious);
    }
}
```

Obrázek 30 Cyklus v modulu B [zdroj vlastní]

Ve for cyklu jsou vytvořeny dvě instance třídy SignatureSetModel. Jeden model slouží pro načtení signatur aktuálně zpracovávané aplikace ze zařízení, druhý je pro signatury z databáze. První model je naplněn po ověření, jestli se nejedná o software pro detekci malwaru. Druhý model je plněn v dalším for cyklu. Cyklus prochází všechny záznamy v databázi a pomocí metody *get()* volané na seznamu načítá jednotlivé signatury k dalšímu zpracování v Modulu C (Obr. 30).

5.2.4 Modul C – Detekce a odstranění malware

Modul C je umístěn v modulu B. To znamená, že stejně jako předchozí část je i tato aplikační logika součástí kódu, který se spustí po stisku tlačítka Malware Scan. Modul B plní dvě funkce: detekci malwaru a jeho následné smazání. Detekce vyžaduje načtení signatur aktuálně zpracovávané aplikace a také signatur z databáze, které se využívají pro porovnání. Kvůli těmto skutečnostem je modul C vložen do modulu B. Informace jsou načteny v modulu B přesně pro potřeby detekčního systému. Metoda *isMalware()* pro vyhodnocení malwaru přijímá jako parametry aplikaci ze zařízení, otisk signatur aplikace z databáze a List<String>, do kterého je přidán název infikovaného balíčku. Název balíčku, který se přidá do seznamu metodou *add()* je následně využit ve druhé části modulu, kdy jsou aplikace na základě jejich package name označené jako malware a smazány.

5.2.4.1 Detekce malware

Pro detekci malware je implementována metoda *isMalware()*. Ta se volá ve for cyklu, který byl popsán v sekci Modul B (Obr. 30). Vyhodnocení aplikace probíhá v tomto cyklu, protože se zde načítají signatury jednotlivých aplikací a je to méně náročné, než vytvářet další seznam signatur a cyklus.

Vyhodnocení aplikace probíhá na základě sesbíraných signatur. Metoda *isMalware()* je složena z několika podmínek. Metody pro klasifikaci jednotlivých signatur mají návratovou boolean – (pravda, nepravda), proto je detekční metoda *isMalware()* složena z podmínek – v případě shody signatur je vrácena hodnota true a package name infikované aplikace je uložen do globální proměnné typu List<String>. Pokud není nalezena shoda vyhodnotí se další signatury aplikací.

Certifikát: Detekce signatur začíná porovnáním MD5 certifikátů. Haš je vyhodnocena jako první, protože se jedná pouze o krátký textový řetězec, takže jeho zpracování je velmi rychlé. Metoda *classifyMD5()* stejně jako další detekční funkce přijímá jako parametry aplikace ze

zařízení a databáze. Její návratová hodnota je boolean. Pomocí porovnání Stringů se zjistí shoda mezi certifikáty SignatureSetModelů. Při nalezení shody je detekční algoritmus ukončen a název aplikace je uložen do seznamu infikovaných aplikací.

Package name: Pokud není nalezena shoda v certifikátech aplikací pokračuje se v kontrole názvů balíčků. Tato kontrola je umístěna jako druhá, protože také porovnává pouze textové řetězce. Princip je analogický jako u certifikátu.

Permissions a uses-features: V případě, selhání detekce certifikátu a názvu balíčku následuje kontrola oprávnění, která je spojena i s ověřením uses-feature tagů. Je to z důvodů, které jsou uvedeny v oddíle 4.3 Navrhnutí příznaků použitelných pro klasifikaci malwaru. Podstata je stále stejná - hledání shody v obou signaturách. Namísto metody *equals()* používané při srovnání textových řetězců je však implementována metoda *equalLists()*. Jelikož oprávnění je možné napsat v libovolném pořadí je nutné je nejprve seřadit. Seřazení obou seznamů je realizováno s využitím datového typu Collections, který umožňuje, jak seřadit, tak porovnat seznamy oprávnění. Se seznamy lze pracovat jen v tom případě, že nejsou prázdné a mají stejnou velikost. Tyto problémy řeší metoda *equalLists()*. Stejně se přistupuje k vyhodnocení uses-features, jelikož jsou tato data ve stejném formátu.

Komponenty aplikace: Klasifikace komponent kontroluje strukturu aplikace. Pokud se neshodují počty všech komponent (aktivit, služeb, providerů a receiverů), je klasifikace ukončena a aplikace není vyhodnocena jako malware. Tato kontrola je poslední, jelikož může při shodě obsáhlých programů trvat velmi dlouho a vzhledem k jednoduchým možnostem přejmenování komponent aplikace nemusí být až tak efektivní. Pokud by měla aplikace změněny oprávnění a ikonu, pak se změní i výstupní hodnota MD5. Škodlivý program by se tímto mohl před detekčním algoritmem skrýt a následná klasifikace komponent se nabízí jako možné řešení. Metoda nejprve ověří shodné velikosti seznamů komponent a následně porovná shody v jejich názvech. Při shodě je aplikace vyhodnocena jako malware, jinak je detekční algoritmus ukončen (Obr. 31).

```
public boolean classifyComponents(SignatureSetModel deviceApp, SignatureSetModel dbApp){

    boolean isSameRec,isSameAct,isSameServ, isSameProv;
    if(deviceApp.activityNames.size() == dbApp.activityNames.size()
            && deviceApp.servicesNames.size() == dbApp.servicesNames.size()
            &&  deviceApp.providersNames.size() == dbApp.providersNames.size()
            && deviceApp.receiversNames.size() == dbApp.receiversNames.size()){

        isSameAct = equalLists(deviceApp.activityNames,dbApp.activityNames);
        isSameServ = equalLists(deviceApp.servicesNames,dbApp.servicesNames);
        isSameProv = equalLists(deviceApp.providersNames,dbApp.providersNames);
        isSameRec = equalLists(deviceApp.receiversNames,dbApp.receiversNames);

        return isSameAct && isSameRec && isSameProv && isSameServ;
        }else
            return false;

}
```

Obrázek 31 Klasifikace komponent [zdroj vlastní]

5.2.4.2 *Odstranění malware*

Druhá část modulu C má za úkol detekované aplikace smazat ze zařízení. Tento kód na rozdíl od detekce není součástí modulu B, ale je umístěn v samostatném for cyklu, který prochází seznam aplikací označených detekčním softwarem jako malware. Názvy aplikací označených jako malware jsou předány jako parametr metodě *deleteApp()*, která vyzve uživatele k odinstalovaní zpracovávané aplikace.

```
public void deleteApp(String packageName){

    Intent intent = new Intent(Intent.ACTION_DELETE);
    intent.setData(Uri.parse("package:"+packageName));
    startActivity(intent);

}
```

Obrázek 32 Mazání aplikací [zdroj vlastní]

Metoda využívá objektu intent, který má dvě části – action a data. Action část obsahuje popis akce, která bude provedena v tomto případě ACTION_DELETE. Akce smaže data z datové části intentu, data zde znamenají název balíčku, který byl označen jako malware. Posledním krokem ke smazání aplikace z programátorského hlediska je použití metody *startActivity()*, která má jako parametr právě vytvořený intent (Obr. 32). Z uživatelského hlediska to však

ještě neznamená odstranění aplikace, nýbrž pouze zobrazení dialogového okna s výzvou ke smazání problémové aplikace.

5.2.5 Další části detekční aplikace

Refresh button: Tlačítko sloužilo původně pro obnovu smazaných aplikací v grafickém uživatelském rozhraní. V době, kdy je zobrazena výzva ke smazání aplikace je již většinou dokončena aplikační logika tlačítka Malware Scan. To znamená, že se aktualizují hodnoty v GUI, ale počet odinstalovaných aplikací neodpovídá skutečnému stavu. Proto je třeba ověřit, zda byly aplikace označené jako malware úspěšně smazány. Po stisknutí refresh tlačítka se metodou *verifyAppIsInstalled()* ověří, jestli názvy balíčku, které měly být smazány jsou dostupné na zařízení, nebo ne. Následně vypíše aktuální počet smazaných aplikací do příslušného textového pole. Analogicky se pracuje i s dalšími částmi GUI, takže tlačítko pomocí globálních proměnných v programu aktualizuje data v uživatelském rozhraní. Navíc toto tlačítko ošetřuje neaktuální hodnoty, které vznikaly použitím globálních proměnných k výpisu. Například počet infikovaných aplikací v zařízení se rovná počtu nalezených malwarů mínus počet smazaných malwarů. Toto má za následek výpis záporné hodnoty při nulovém počtu malwarů a zároveň více, než jednom smazaném malwaru. Právě tyto typy problémů řeší aktualizační tlačítko (Obr. 33). V produkční aplikaci by tato logika byla řešena například v metodě *onResume()*, v prototypu tlačítko slouží pro testování a prezentaci aktuálních informací o aplikacích na zařízení.

```
btn_refresh.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        PackageManager pm = getPackageManager();
        List<String> deleted = new ArrayList<>();
        int newMalicious;
        for(int i = 0; i < toDelete.size(); i++){

            final String packageToDelete = toDelete.get(i);
            if(!verifyAppIsInstalled(packageToDelete, pm)){
                deleted.add(packageToDelete);
            }

        }
        if(malicious.size()>0){
            newMalicious = malicious.size() - deleted.size();
        }
        else {
            newMalicious = malicious.size();
        }
        tv_maliciousAppCount.setText(String.valueOf(newMalicious));
        tv_appCountid.setText(String.valueOf(GetNonSystemApks().size()));
        tv_deletedAppsCount.setText(String.valueOf(deleted.size()));
        if(deleted.size() >0){
            malicious = new ArrayList<>();
        }
    }
});
```

Obrázek 33 Aktualizační tlačítko [zdroj vlastní]

Načítání databáze z assetů: Funkcionalita načítání databáze ze složky assets slouží zejména v produkčních aplikacích, které potřebují pro správný běh databázi. V tomto projektu to je možné po vytvoření nového virtuálního zařízení (Android Virtual Device). Díky této funkcionalitě lze vytvořit jakýkoliv emulátor s API 30 (a vyšší). Na tomto emulátoru pomocí aplikace Detector umožňuje detekovat známý malware na základě signatur, které jsou uloženy v databázi. Zajímavostí zde je, že na rozdíl od klasické Javy, kde pro načtení databáze stačí načíst *.db soubor je v operačním systému Android nutné načíst i soubor *.db-journal. V případě, že tento soubor není načten spolu s databází je sice načtena databáze a software vypadá, že pracuje správně. Nicméně ve skutečnosti vytvoří Android OS svůj journal soubor pro databáze. Ta je ovšem přepsána a je prázdná, takže se nepodaří detekovat žádnou aplikaci jako malware. Uvedený problém byl vyřešen přidáním souboru journal do

adresáře assets a jeho načtením stejným způsobem jako byla načtena databáze (Obr. 34). Součástí funkcionality je i ověření, zda již databáze existuje, nebo ne.

```
Context context = getApplicationContext();
String appDataPath = context.getApplicationInfo().dataDir;

File dbFolder = new File( pathname: appDataPath + "/databases");
dbFolder.mkdir();//This can be called multiple times.

File dbFilePath = new File( pathname: appDataPath + "/databases/malware.db");
File journalFilePath = new File( pathname: appDataPath + "/databases/malware.db-journal");

try {
    InputStream inputStream = context.getAssets().open( fileName: "malware.db");
    OutputStream outputStream = new FileOutputStream(dbFilePath);
    byte[] buffer = new byte[1024];
    int length;
    while ((length = inputStream.read(buffer))>0)
    {
        outputStream.write(buffer, off: 0, length);
        Log.e( tag: "DBout", msg: "loaded");
    }
    outputStream.flush();
    outputStream.close();
    inputStream.close();
} catch (IOException e){
    //handle
    Log.e( tag: "DBout", msg: "not loaded");
}
```

Obrázek 34 Načítání souboru malware.db [zdroj vlastní]

5.3 Testování

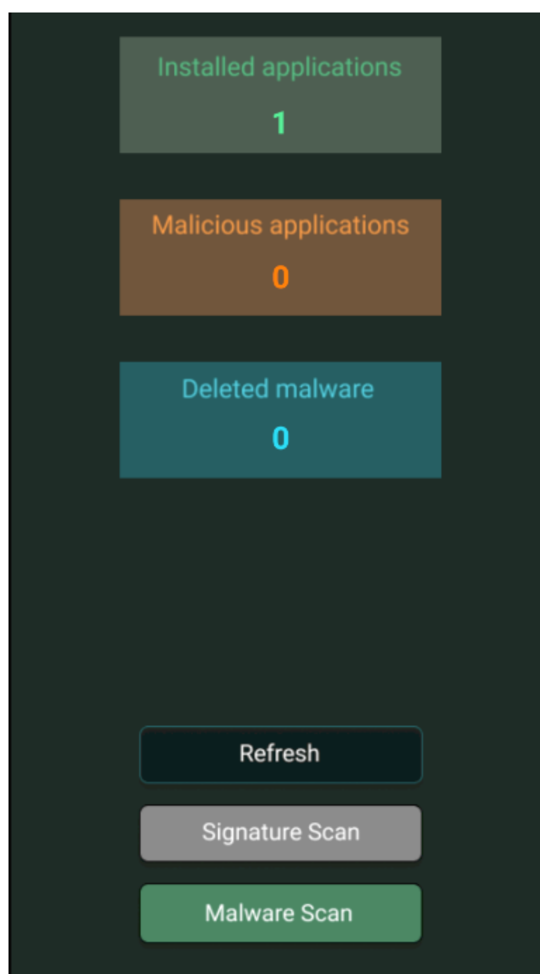
Testování detekčního softwaru probíhalo ve vývojovém prostředí Android Studio v emulátorech (API 30). Po vytvoření logiky pro detekci malwaru bylo využito virtuální zařízení (emulátor), pomocí kterého byla vytvořena databáze vzorků malwaru popsaná v sekci Modul A. Celkem bylo nainstalováno 107 vzorků malwaru. Virtuální zařízení bylo zapnuto a byla v něm spuštěna aplikace Detector. Na základě sesbíraných signatur se podařilo všech 107 nainstalovaných vzorků malwaru ze zařízení odinstalovat.

Výše uvedené testování proběhlo na základě certifikátů. Aby se zajistilo, že správně pracují i ostatní části detekce bylo nutné při testování jednotlivé stupně ochrany dočasně odstranit a ověřit jejich správné fungování pomocí logování do konzole. Tato část testování byla rovněž úspěšná a malware se podařilo detekovat ve všech stupních ochrany. Problém nastal u detekce na základě názvu balíčku v případu, kdy malware změní package name na náhodnou hodnotu. Avšak i malware, který změnil package name, byl rozpoznán a to na základě

oprávnění a dalších stupňů ochrany včetně testu komponent. Součástí testování bylo i zavedení falešných vzorků do databáze, kdy bylo několika vzorkům uměle změněna signatura (simulace obfuskování), tyto malwary se také podařilo detekovat a následně byly z databáze smazány.

5.4 Verzování

Ke správě verzí aplikace byl využíván nástroj GitHub Desktop, který poskytuje grafické prostředí ke cloudovému úložišti GitHub. Detekční program, který byl v rámci bakalářské práce vyvíjen, měl 10 hlavních verzí. Verzování sloužilo jak k zálohování pro případ ztráty nebo poškození dat, tak pro případ, kdy by bylo nutného se vrátit k předešlé verzi. Verze se liší buď logikou, nebo grafickým prostředím. Poslední verze nabízí aktualizované grafické prostředí ve kterém je původní prototypové tlačítko refresh nahrazeno tlačítkem, které odpovídá stylu aplikace (Obr. 35).



Obrázek 35 Finální GUI [zdroj vlastní]

ZÁVĚR

Tvůrci malwaru ať už s motivací finanční, nebo motivací získat data, hledají stále nové zranitelnosti. Proto je velmi důležité, aby uživatelé svá zařízení chránili nejen častými aktualizacemi, ale také svým zodpovědným přístupem (neinstalovat neoficiální a podezřelé aplikace, nenavštěvovat problematické webové stránky apod.) a v neposlední řadě, aby rovněž využívali software pro detekci malwaru.

V práci byl popsán operační systém Android, jeho struktura a aplikace. Podařilo se popsat komponenty aplikací a jejich zranitelná místa (např. nutnost ukončování nevázaných služeb). Díky tomu mohla být provedena úspěšná analýza vzorků malware a její vyhodnocení. Na základě tohoto vyhodnocení byly navrženy netriviální příznaky, které je možné využít pro klasifikaci aplikace jako malware. Dále se podařilo specifikovat druhy detekcí, které se na platformě Android OS využívají.

Detekční aplikace je stoprocentně spolehlivá, co se týče známých vzorků. Výhodou této detekce je, že do jisté míry umí reagovat na obfuskační techniky díky několikastupňové ochraně. Detekce je dost rychlá na to, aby uživatele neodradila od používání aplikace. Prototyp aplikace se velmi blíží reálným komerčním systémům a to způsobem vyhodnocení malwaru. Detekce pracuje na základě pattern matchingu a shoda je vyhodnocena v několika vrstvách. Všechny vzorky malwaru, které se podařilo nainstalovat na Android Virtual Device (API 30) byly úspěšně detekovány. Součástí vyhodnocení aplikace je i možnost smazání infikované aplikace. O všech potřebných datech informuje GUI, které se velmi podobá produkčním malware scannerům.

V práci je vyřešena interakce aplikace s databází, která je načítána z assetů aplikace jako u produkčních aplikací. Díky tomu by bylo možné pracovat s databází nezávisle na zařízení Android OS. Aplikace reaguje i na problémy s ukládáním složených datových typů v databázi SQLite a to použitím formátu JSON.

Práce reaguje na moderní malware technikou pattern matching s vysokou úspěšností. Podařilo se implementovat funkční anti-malware detekční aplikaci s přehledným GUI a uspokojivým výsledkem vyhodnocení malwaru.

6 OMEZENÍ PRÁCE A DALŠÍ MOŽNÝ VÝVOJ

Nejvíce limitujícím faktorem prototypu aplikace je velikost databáze. Její rozšíření je důležitým krokem do budoucna pro detekci co největšího možného množství malwaru.

Množství detekovaného malwaru je omezeno vyhodnocováním na základě pravdivostní logiky. To má za následek selhání detekce při shodě například 99 procent. Reakcí na tuto skutečnost v budoucnosti bude implementace fuzzy logiky, která nepracuje jen s hodnotami pravda, nepravda. Fuzzy logika je ideálním řešením, jelikož může nabýt jakékoliv hodnoty z intervalu <0;1>. Reakcí na tuto aplikační logiku by bylo implementování metody, která by v případě vysoké pravděpodobnosti, že se jedná o malware (P blížící se k 1) zaslala podezřelou aplikaci k manuální analýze na vyhodnocení.

Jako další vylepšení se nabízí reakce na obfuskované aplikace. To je do určité míry možné i nyní, ale ideálním řešením by bylo přidání dalšího stupně ochrany. Tato ochrana by zaznamenávala sekvence volání API, ze které by bylo možné extrahovat jen podezřelé sekvence a tyto následně zahrnout do porovnávání.

Jako omezení se jeví vyhodnocení aplikace na základě package name. Tato ochrana je součástí zadání, nicméně její možnosti jsou omezené. Neboť analýza provedená v rámci této práce odhalila, že některé vzorky malwaru zneužívají jména legitimních aplikací. Řešení se nabízí v podobě vytvoření white-listu legitimních aplikací, ve kterém by byly zahrnuty i ty systémové.

Plánovaným vylepšením do budoucna je přechod z prototypu aplikace do produkčního prostředí. To by mimo zveřejnění aplikace zahrnovalo následující úpravy GUI: nahrazení refresh tlačítka logikou v *onResume()*; přidání recycler view s detailními informacemi o aplikacích na mobilním zařízení; pokročilí uživatelé by měli možnost tlačítkem Signature Scan přidávat vzorky do databáze signatur.

Jako nejlepší vylepšení se do budoucna jeví využití neuronových sítí a umělé inteligence. Systém je pak schopen se učit na základě dat a poté se i rozhodovat. Výhodou je, že poté, co se nastaví váhy, již nejsou potřebné tak velké databáze jako u pattern matchingu. Takže jsou méně náročné, co se úložiště týče.

SEZNAM POUŽITÉ LITERATURY

[1] MUELLER, Bernhard, Sven SCHLEIER, Jeroen WILLEMSEN a Carlos HOLGUERA. OWASP. Mobile Security Testing Guide: Introduction to the Mobile Security Testing Guide. *OWASP* [online]. 12 May 2020 [cit. 2020-05-25]. Dostupné z: https://mobile-security.gitbook.io/mobile-security-testing-guide/.

[2] VERMA, Prashant a Akshay DIXIT. *Mobile Device Exploitation Cookbook*. Birmingham: Packt Publishing, 2016. ISBN 1783558725.

[3] O'DEA, S. *Smartphone users worldwide 2016-2021* [online]. 28 Feb 2020 [cit. 2020-05-25]. Dostupné z: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[4] PURPLESEC LLC. *2019 Cyber security statistics and trends* [online]. c2020 [cit. 2020-05-25]. Dostupné z: https://purplesec.us/resources/cyber-security-statistics/?fbclid=IwAR3G34zdhijjH3WziJ8eUXOTgjHQYJmSXpuX51OERE6A3Y_ELocfWx5cu54.

[5] KEMP, Simon. *DIGITAL 2019: Q2 GLOBAL DIGITAL STATSHOT* [online]. In: 25 APRIL 2019 [cit. 2020-05-25]. Dostupné z: https://datareportal.com/reports/digital-2019-q2-global-digital-statshot?rq=android%20.

[6] STATCOUNTER. *Mobile Operating System Market Share Worldwide* [online]. April 2020 [cit. 2020-05-26]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide.

[7] STATCOUNTER. *Desktop vs Mobile vs Tablet vs Console Market Share Worldwide* [online]. April 2020 [cit. 2020-05-26]. Dostupné z: https://gs.statcounter.com/platform-market-share#monthly-201901-202004.

[8] MUELLER, Bernhard, Sven SCHLEIER, Jeroen WILLEMSEN a Carlos HOLGUERA. OWASP. Mobile Security Testing Guide: Android Testing Guide. *OWASP* [online]. 12 May 2020 [cit. 2020-05-25]. Dostupné z: https://mobile-security.gitbook.io/mobile-security-testing-guide/.

[9] JUHAŇÁK, Petr. Android hacking pro začátečníky – architektura. *Hackingkurzy* [online]. České Meziříčí: Hackerlab, 2015, 25. 02. 2018 [cit. 2020-05-31]. Dostupné z: https://www.hackingkurzy.cz/blog/android-hacking-pro-zacatecnik-arhitektura/.

[10] GOOGLE A.S. Android Developer: Platform Overview. *Android Developer Website* [online]. [cit. 2020-05-02]. Dostupné z: https://developer.android.com/guide/platform.

[11] SANZ, Borja, Igor SANTOS, Carlos LAORDEN, Xabier UGARTE-PEDRERO, Javier NIEVES, Pablo G. BRINGAS a Gonzalo ÁLVAREZ MARAÑÓN. MAMA: MANIFEST ANALYSIS FOR MALWARE DETECTION IN ANDROID. *Cybernetics and Systems* [online]. 2013, **44**(6-7), 469-488 [cit. 2020-08-08]. DOI: 10.1080/01969722.2013.803889. ISSN 0196-9722. Dostupné z: http://www.tandfonline.com/doi/abs/10.1080/01969722.2013.803889.

[12] SADIQ, Ayesha, Yuan-Fang LI a Sea LING. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software* [online]. 2020, **159** [cit. 2020-08-08]. DOI: 10.1016/j.jss.2019.110450. ISSN 01641212. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S0164121219302249 .

[13] GOOGLE. Android Developer: App Manifest Overview. *Android Developer Website* [online]. [cit. 2020-04-15]. Dostupné z: https://developer.android.com/guide/topics/manifest/manifest-intro.

[14] GOOGLE A.S. Android Developer: Reduce your app size. *Android Developer Website* [online]. [cit. 2020-04-15]. Dostupné z: https://developer.android.com/topic/performance/reduce-apk-size.

[15] QIU, Junyang, Wei LUO, Lei PAN, Yonghang TAI, Jun ZHANG a Yang XIANG. Predicting the Impact of Android Malicious Samples via Machine Learning. *IEEE Access* [online]. 2019, **7**, 66304-66316 [cit. 2020-08-08]. DOI: 10.1109/ACCESS.2019.2914311. ISSN 2169-3536. Dostupné z: https://ieeexplore.ieee.org/document/8703863/GUARDSQUARE.

[16] Optimizing Android resources. In: *Guardsquare.com* [online]. Leuven: Guardsquare nv, 2016, 19 June 2018 [cit. 2020-04-21]. Dostupné z: https://www.guardsquare.com/en/blog/optimizing-android-resources.

[17] GOOGLE A.S. Android Developer: Asset Manager. *Android Developer Website* [online]. [cit. 2020-04-17]. Dostupné z: https://developer.android.com/reference/android/content/res/AssetManager.

[18] GOOGLE. Android Developer: Introduction to Activities. *Android Developer Website* [online]. Mountain View, Kalifornie: Google [cit. 2020-04-18]. Dostupné z: https://developer.android.com/guide/components/activities/intro-activities.

[19] GOOGLE A.S. Android Developer Documentation: Understand the Activity Lifecycle. *Android Developer Website: Activity Lifecycle* [online]. Mountain View, Kalifornie: Google [cit. 2020-04-18]. Dostupné z: https://developer.android.com/guide/components/activities/activity-lifecycle.

[20] ARMANDO, Alessandro, Roberto CARBONE, Gabriele COSTA a Alessio MERLO. Android Permissions Unleashed. In: *2015 IEEE 28th Computer Security Foundations Symposium* [online]. IEEE, 2015, 2015, s. 320-333 [cit. 2020-08-08]. DOI: 10.1109/CSF.2015.29. ISBN 978-1-4673-7538-2. Dostupné z: https://ieeexplore.ieee.org/document/7243742/.

[21] GOOGLE A.S. Android Developer Documentation: Permissions overview. *Android Developer Website* [online]. Mountain View, Kalifornie: Google [cit. 2020-04-18]. Dostupné z: https://developer.android.com/guide/topics/permissions/overview.

[22] GOOGLE A.S. Android Developer Documentation: Services Overview. *Android Developer Website: Services Overview* [online]. Mountain View, Kalifornie: Google [cit. 2020-04-11]. Dostupné z: https://developer.android.com/guide/components/services.

[23] KOMATINENI, S a D MACLEAN. *Understanding Content Providers* [online]. Pro Android 4: Apress, 2012 [cit. 2020-05-10]. DOI: https://doi.org/10.1007/978-1-4302-3931-4_4. Dostupné z: https://link.springer.com/chapter/10.1007%2F978-1-4302-3931-4_4.

[24] GOOGLE A.S. Android Developer Documentation: Broadcasts overview. *Android Developer Website* [online]. Mountain View, Kalifornie: Google [cit. 2020-05-9]. Dostupné z: https://developer.android.com/guide/components/broadcasts.

[25] GOOGLE A.S. Android Developer Documentation: Security tips – Store data. *Android Developer Website* [online]. Mountain View, Kalifornie: Google [cit. 2020-05-9]. Dostupné z: https://developer.android.com/training/articles/security-tips#StoringData.

[26] GOOGLE A.S. Android Developer Documentation: Intents and Intent Filters. *Android Developer Website* [online]. Mountain View, Kalifornie: Google [cit. 2020-05-9]. Dostupné z: https://developer.android.com/guide/components/intents-filters.

[27] GOOGLE A.S. Android Developer Documentation: APK Signature Scheme v2. *Android Developer Website* [online]. Mountain View, Kalifornie: Google [cit. 2020-05-9]. Dostupné z: https://source.android.com/security/apksigning/v2.

[28] New Android P includes several security improvements. *Malwarebytes* [online]. Santa Clara: Malwarebytes, c2020 [cit. 2020-06-21]. Dostupné z: https://blog.malwarebytes.com/cybercrime/2018/07/android-p-security-improvements/.

[29] Encyclopedia: History of Malicious programs. *Encyclopedia by Kaspersky* [online]. Moskva: Kaspersky Lab, c2020 [cit. 2020-06-25]. Dostupné z: https://encyclopedia.kaspersky.com/knowledge/history-of-malicious-programs/.

[30] North Korean Malicious Cyber Activity. *Cybersecurity and Infrastructure Security Agency (CISA)* [online]. Rosslyn: Cybersecurity and Infrastructure Security Agency (CISA) [cit. 2020-06-25]. Dostupné z: https://www.us-cert.gov/northkorea.

[31] Malware vs Viruses. *McAfee* [online]. Santa Clara: McAfee, c2020 [cit. 2020-06-25]. Dostupné z: https://www.mcafee.com/enterprise/en-us/security-awareness/ransomware/malware-vs-viruses.html.

[32] Malware. *Eset* [online]. Bratislava: ESET, spol. s r.o., c1992 – 2020 [cit. 2020-06-25]. Dostupné z: https://www.eset.com/cz/malware/.

[33] JIANG, Xuxian a Yajin ZHOU. *Android malware* [online]. Raleigh: Springer, 2013. ISBN 978-1461473930.

[34] MERCALDO, Francesco, Vittoria NARDONE, Antonella SANTONE a Corrado Aaron VISAGGIO. Download malware? no, thanks. In: *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering - FormaliSE '16* [online]. New York, New York, USA: ACM Press, 2016, 2016, s. 22-28 [cit. 2020-08-08]. DOI: 10.1145/2897667.2897673. ISBN 9781450341592. Dostupné z: http://dl.acm.org/citation.cfm?doid=2897667.2897673.

[35] What Is a Drive-By Download? *Kaspersky* [online]. Moskva: Kaspersky Lab, c2020 [cit. 2020-06-27]. Dostupné z: https://www.kaspersky.com/resource-center/definitions/drive-by-download.

[36] YANG, Yaping, Lizhi CAI a Yanguo ZHANG. Research on non-authorized privilege escalation detection of android applications. In: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* [online]. IEEE, 2016, 2016, s. 563-568 [cit. 2020-08-08]. DOI: 10.1109/SNPD.2016.7515959. ISBN 978-1-5090-2239-7. Dostupné z: http://ieeexplore.ieee.org/document/7515959/.

[37] J. Alegre, J.-C. Cortes, F.-J. Santonja, and R.-J. Villanueva, "Quantifying the behaviour of the actors in the spread of Android malware infection," *in Mathematical Modeling in SocialSciences and Engineering*, Nova Science Publishers, New York, NY, USA, 2013 *(2) (PDF) Agent-Based Model to Study and Quantify the Evolution Dynamics of Android Malware Infection*. Available from: https://www.researchgate.net/publication/285476039_Agent-Based_Model_to_Study_and_Quantify_the_Evolution_Dynamics_of_Android_Malware_Infection [accessed Aug 08 2020].

[38] Command and Control [C&C] Server. *Trend Micro* [online]. Tokio: Trend Micro Incorporated, c2020 [cit. 2020-06-29]. Dostupné z: https://www.trendmicro.com/vinfo/us/security/definition/command-and-control-server.

[39] QUIRCHMAYR, Gerald, Josef BASL, Ilsun YOU, Lida XU a Edgar WEIPPL, ed. *Multidisciplinary Research and Practice for Information Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012 [cit. 2020-08-08]. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-32498-7. ISBN 978-3-642-32497-0.

[40] CHOUDHARY, Mahima a Brij KISHORE. HAAMD: Hybrid Analysis for Android Malware Detection. In: *2018 International Conference on Computer Communication and Informatics (ICCCI)* [online]. IEEE, 2018, 2018, s. 1-4 [cit. 2020-08-08]. DOI: 10.1109/ICCCI.2018.8441295. ISBN 978-1-5386-2238-4. Dostupné z: https://ieeexplore.ieee.org/document/8441295/.

[41] BAKOUR, Khaled, H. Murat UNVER a Razan GHANEM. The Android Malware Static Analysis: Techniques, Limitations, and Open Challenges. In: *2018 3rd*

International Conference on Computer Science and Engineering (UBMK) [online]. IEEE, 2018, 2018, s. 586-593 [cit. 2020-08-08]. DOI: 10.1109/UBMK.2018.8566573. ISBN 978-1-5386-7893-0. Dostupné z: https://ieeexplore.ieee.org/document/8566573/.

[42] MIRZAEI, O., J.M. DE FUENTES, J. TAPIADOR a L. GONZALEZ-MANZANO. AndrODet: An adaptive Android obfuscation detector. *Future Generation Computer Systems* [online]. 2019, **90**, 240-261 [cit. 2020-08-08]. DOI: 10.1016/j.future.2018.07.066. ISSN 0167739X. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S0167739X18309312.

[43] GĄSIENIEC, Leszek a Frank WOLTER, ed. *Fundamentals of Computation Theory* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013 [cit. 2020-08-08]. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-40164-0. ISBN 978-3-642-40163-3.

[44] What is Code Obfuscation? *Preemptive* [online]. Mayfield Village: PreEmptive, c2020 [cit. 2020-06-30]. Dostupné z: https://www.preemptive.com/obfuscation?utm_source=google&utm_medium=cpc&utm_campaign=obfuscation&gclid=CjwKCAjwxev3BRBBEiwAiB_PWKD7vVUuRH0C9gzBlZjA-9H2-nk4mhYFw7rysynMliczwm02qvLldBoCx84QAvD_BwE.

[45] MORALES, Jose Andre, Ravi SANDHU a SHOUHUAI XU. Evaluating detection and treatment effectiveness of commercial anti-malware programs. In: *2010 5th International Conference on Malicious and Unwanted Software* [online]. IEEE, 2010, 2010, s. 31-38 [cit. 2020-08-08]. DOI: 10.1109/MALWARE.2010.5665797. ISBN 978-1-4244-9353-1. Dostupné z: http://ieeexplore.ieee.org/document/5665797/.

[46] Abraham Rodríguez-Mota, Ponciano J. Escamilla-Ambrosio and Moisés Salinas-Rosales (November 2nd 2017). Malware Analysis and Detection on Android: The Big Challenge, Smartphones from an Applied Research Perspective, Nawaz Mohamudally, IntechOpen, DOI: 10.5772/intechopen.69695. Dostupné z: https://www.intechopen.com/books/smartphones-from-an-applied-research-perspective/malware-analysis-and-detection-on-android-the-big-challenge.

[47] TAHIR, Rabia. A Study on Malware and Malware Detection Techniques. *International Journal of Education and Management Engineering* [online]. 2018, **8**(2), 20-30 [cit. 2020-08-08]. DOI: 10.5815/ijeme.2018.02.03. ISSN 23053623. Dostupné z: http://www.mecs-press.org/ijeme/ijeme-v8-n2/v8n2-3.html.

[48] ZHENG, Min, Mingshen SUN a John C.S. LUI. Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* [online]. IEEE, 2013, 2013, s. 163-171 [cit. 2020-08-08]. DOI: 10.1109/TrustCom.2013.25. ISBN 978-0-7695-5022-0. Dostupné z: http://ieeexplore.ieee.org/document/6680837/.

[49] JIAMIN BAO, XIAOHUI YANG, TAO LI a JINXIN ZHANG. A detection system of android application based on permission analysis. In: *2014 Communications Security Conference (CSC 2014)* [online]. Institution of Engineering and Technology, 2014, 2014, s. 5-5 [cit. 2020-08-08]. DOI: 10.1049/cp.2014.0730. ISBN 978-1-84919-844-8. Dostupné z: https://digital-library.theiet.org/content/conferences/10.1049/cp.2014.0730.

[50] SATO, Ryo, Daiki CHIBA a Shigeki GOTO. Detecting Android Malware by Analyzing Manifest Files. *Proceedings of the Asia-Pacific Advanced Network* [online]. 2013, **36**, 23-31 [cit. 2020-08-08]. DOI: 10.7125/APAN.36.4. ISSN 2227-3026. Dostupné z: http://journals.sfu.ca/apan/index.php/apan/article/view/110.

[51] CHAN, Patrick P. K. a WEN-KAI SONG. Static detection of Android malware by using permissions and API calls. In: *2014 International Conference on Machine Learning and Cybernetics* [online]. IEEE, 2014, 2014, s. 82-87 [cit. 2020-08-08]. DOI: 10.1109/ICMLC.2014.7009096. ISBN 978-1-4799-4215-2. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7009096.

[52] LI, Qi a Xiaoyu LI. Android Malware Detection Based on Static Analysis of Characteristic Tree. In: *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery* [online]. IEEE, 2015, 2015, s. 84-91 [cit. 2020-08-08]. DOI: 10.1109/CyberC.2015.88. ISBN 978-1-4673-9200-6. Dostupné z: http://ieeexplore.ieee.org/document/7307791/.

[53] KANG, Hyunjae, Jae-wook JANG, Aziz MOHAISEN a Huy Kang KIM. Detecting and Classifying Android Malware Using Static Analysis along with Creator Information. *International Journal of Distributed Sensor Networks* [online]. 2015, **11**(6) [cit. 2020-08-08]. DOI: 10.1155/2015/479174. ISSN 1550-1477. Dostupné z: http://journals.sagepub.com/doi/10.1155/2015/479174.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

HAL Hardware Abstract Layer – Abstraktní hardwarová vrstva

APK Formát souboru využívaný pro distribuci a instalaci softwaru na Android zařízení

HW Hardware – fyzické vybavení výpočetního stroje

SW Software – systémové vybavení výpočetního stroje

ART Android Runtime – běhové prostředí Android

API Application Programming Interface – Programovací rozhraní aplikace

VM Virtual Machine – virtuální zařízení

DEX Dalvik executable – formát souboru pro Dalvik Virtual Machine

XML Extensible Markup Language – značkovací jazyk

UML Unified Modeling Language – modelovací jazyk

REST Representational State Transfer – architektura rozhraní využívající http volání

URI Uniform Resource Indetifier – textový řetězec označující původ zdroje

IPC Inter Process Communication – způsob komunikace mezi procesy

AES Advanced Encrypting Standard – algoritmus k šifrování dat

DES Data Encryption Standard – algoritmus k šifrování dat

CFG Control Flow Graph – grafické znázornění

IDE Integrated Development Environment – vývojové prostředí

GUI Graphic User Interface – grafické uživatelské rozhraní

SEZNAM OBRÁZKŮ

Obrázek 1 Architektura Android OS [10] ... 12
Obrázek 2 Java vs Dalvik [9] ... 14
Obrázek 3 Struktura APK [12] ... 15
Obrázek 4 Ukázka AndroidManifest.xml [zdroj vlastní] ... 16
Obrázek 5 Deklarace aktivity [zdroj vlastní] ... 17
Obrázek 6 Životní cyklus aktivity [19] ... 18
Obrázek 7 Životní cyklus služeb [22] ... 20
Obrázek 8 APK soubor před a po vložení podepisovacího bloku [27] ... 23
Obrázek 9 Validační proces [1] ... 24
Obrázek 11 Total Commander – žádost o heslo [zdroj vlastní] ... 37
Obrázek 10 Správce souborů Windows - chybová hláška [zdroj vlastní] ... 37
Obrázek 12 Použití nástroje ApkTool [zdroj vlastní] ... 38
Obrázek 13 Použití nástroje Dex2Jar [zdroj vlastní] ... 39
Obrázek 14 JD-GUI [zdroj vlastní] ... 39
Obrázek 15 Použití ApkTool se zachováním dex souborů [zdroj vlastní] ... 41
Obrázek 16 Nová složka s dex souborem [zdroj vlastní] ... 42
Obrázek 17 MobSF – Základní informace [zdroj vlastní] ... 42
Obrázek 18 AndroidManifest.xml [zdroj vlastní] ... 43
Obrázek 19 Schéma detekční části [zdroj vlastní] ... 49
Obrázek 20 Deklarace aktivit [zdroj vlastní] ... 50
Obrázek 21 GUI [zdroj vlastní] ... 51
Obrázek 22 Obsluha tlačítka [zdroj vlastní] ... 52
Obrázek 23 Práce na částech APK [zdroj vlastní] ... 53
Obrázek 24 Konverze pomocí GSON [zdroj vlastní] ... 54
Obrázek 25 Přidání signatury do DB [zdroj vlastní] ... 54
Obrázek 26 Vytvoření databáze [zdroj vlastní] ... 55
Obrázek 27 Přístup ke sloupcům DB [zdroj vlastní] ... 56
Obrázek 28 Metoda *addOne()* [zdroj vlastní] ... 56
Obrázek 29 Načtení seznamu signatur [zdroj vlastní] ... 57
Obrázek 30 Cyklus v modulu B [zdroj vlastní] ... 57
Obrázek 31 Klasifikace komponent [zdroj vlastní] ... 60
Obrázek 32 Mazání aplikací [zdroj vlastní] ... 60
Obrázek 33 Aktualizační tlačítko [zdroj vlastní] ... 62
Obrázek 34 Načítání souboru malware.db [zdroj vlastní] ... 63

Obrázek 35 Finální GUI [zdroj vlastní] 64

SEZNAM PŘÍLOH

Příloha P I: Ohlas bakalářské práce ze společnosti Monet+

Příloha P II: Zdrojový kód

PŘÍLOHA P I: OHLAS BAKALÁŘSKÉ PRÁCE ZE SPOLEČNOSTI MONET+

Zadání bakalářské práce vycházelo z požadavku z praxe na rozšíření funkčnosti stávající malware detekce v produktu společnosti AHEAD iTec, s.r.o. a bylo tudíž ve své podstatě poměrně rozsáhlé.

Na straně řešitele bylo předpokládáno nastudování komplexní problematiky detekce malware na platformě Android, čehož se pan Dorotík zhostil svědomitě a nabyté znalosti promítl zcela samostatně do teoretické části práce. Praktickou část práce a vývoj vlastní detekční funkcionality s námi pan Dorotík pravidelně a proaktivně konzultoval. V rámci těchto konzultací docházelo k vyhodnocení dosud odvedené činnosti, společná diskuze nad možnými vylepšení a vytyčení směru dalšího vývoje. Pan Dorotík domluvené cíle samostatně zpracovával a naprogramoval řešení, které naplňuje parametry zadání práce. Zároveň v práci také reflektoval možný další vývoj, který již byl bohužel časově nad rámec zpracování bakalářské práce.

Se spoluprací s panem Dorotíkem jsme spokojeni a výsledné řešení akceptujeme jako splněné dle zadání.

Za AHEAD iTec
Mgr. Ondřej Gabrhelík, Bc. Ondřej Přikryl, Mgr. Anežka Pejlová

PŘÍLOHA P II: ZDROJOVÝ KÓD APLIKACE

Příloha obsahuje zdrojový kód aplikace, která byla vytvořena. Součástí zdrojového kódu je i databáze příznaků infikovaných aplikací.