Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Android malware-ek memória analízis alapú detektálása

Tudományos Diákköri Konferencia dolgozat

Készítette
Gazdag András

Konzulens
Dr. Buttyán Levente

2014. október 22.

Tartalomjegyzék

Kivonat **4**

Abstract **6**

Bevezető **8**

1. Irodalomkutatás **11**

1.1. PUMA 11
1.2. Rejtőzködő Android rootkit detektálás 12
1.3. Volatility 13
1.4. LiME 13

2. Rendszer felépítése **14**

2.1. UkatemiSHIELD 14
2.2. APKAnalyser 15

3. UkatemiSHIELD **16**

3.1. Specifikáció 16
3.2. A rendszer tervezése 17
3.3. A rendszer működése 18
3.4. Az alkalmazás komponensei 20
3.4.1. ShieldService 20
3.4.2. ShieldTimer 20
3.4.3. ShieldKnowledgeBase 21
3.4.4. DeviceInformationListener 21
3.4.5. ScanManager 21
3.4.6. FileScanner 21
3.4.7. PackageScanner 22
3.5. Az alkalmazást kiszolgáló szolgáltatások 22
3.5.1. Hash API 22
3.5.2. Gépi tanulás API 22

3.6. Az alkalmazás továbbfejlesztése és gyengeségei 23
3.7. Az alkalmazás tesztelése 24
3.7.1. Alapvető funkciók tesztelése 24
3.7.2. Hash API tesztelése 26
3.7.3. Gépi tanulás tesztelése 26

4. APKAnalyser 27
4.1. A rendszer leírása 27
4.1.1. Hozzáférési felületek 27
4.1.2. Analízis környezet 28
4.2. Analízis környezet kialakítása 28
4.2.1. Fizikai eszköz előkészítése 29
4.2.2. Emulátor előkészítése 31
4.2.3. LiME előkészítése 31
4.2.4. Analízis környezet root-olása 32
4.3. Memória tartalom rögzítése 32
4.4. Memóriakép elemzése 32
4.4.1. Folyamatok vizsgálata 33
4.4.2. Hálózati forgalom vizsgálata 33
4.4.3. Fájlrendszerek vizsgálata 34
4.4.4. Rootkit-ek vizsgálata 34
4.5. Automatizált detekció 35
4.5.1. Előkészítés 35
4.5.2. Folyamat vizsgálata 35
4.5.3. Viselkedés meghatározása 37
4.6. Megoldás erősségei és gyengeségei 37

5. Eredmények és Továbbfejlesztés 39
5.1. Vizsgálatok eredménye 39
5.1.1. Malware detekció tesztelése 40
5.1.2. Malware család felismerés tesztelése 41
5.2. Továbbfejlesztési lehetőségek 41

Irodalomjegyzék 44

Függelék 45
F.1. Az UkatemiSHIELD alkalmazás szoftver architektúrája 45
F.2. Leggyakoribb gyanús Android jogosultságok listája 46
F.3. Tiszta Android alkalmazások detekciós értékei 49
F.4. Kártékony Android alkalmazások detekciós értékei 50

F.5. A Lotoor malware család detekciós értékei 51
F.6. A DroidKungFu malware család detekciós értékei 51

Kivonat

Az informatikai biztonsági vizsgálatok és kutatások során több bevett módszert is alkalmaznak a szakértők. Ezek közül az egyik leggyakrabban használt, a nem permanens adattárolók tartalmának vizsgálata (live memory forensics). Az újabb és újabb technológiák megjelenésével folyamatosan szükségessé válik a vizsgálati módszerek fejlesztése és adaptálása. Ennek lehettünk tanúi a mobil platformok területén is az elmúlt években. Dolgozatomban az Android platformra elérhető memória analízist elősegítő eszközökkel foglalkozom. A jelenleg elérhető megoldások segítségével, azok egyesítésével és továbbfejlesztésével, egy automatizált eszközt készítettem, amely egy android platformra készített alkalmazás vizsgálatát végzi el. A szoftver célja, hogy elvégezze azokat a műveleteket előre, amelyek elősegítik és támogatják egy humán szakértő által végzett vizsgálatot. Az analízist egy erre a célra kialakított és a későbbi elemzésekhez előkészített emulált környezetben végzem, ahol lehetséges a potenciálisan ártó szándékú alkalmazások biztonságos vizsgálata. Az elkészült rendszer egy webes szolgáltatásként működik, amihez két ponton lehetséges csatlakozni. A rendszerhez tartozik egy kliens alkalmazás, amely android alapú eszközökön fut, és a rendszerre telepített további alkalmazásokat vizsgálja. Amennyiben egy alkalmazást gyanúsnak ítél meg, akkor annak futtatható állományát elküldi a szerverre, ahol további elemzésekre kerül sor. A másik elérési pontja a szolgáltatásnak egy hagyományos webes felület. Itt bárkinek lehetősége van feltölteni ".apk" formátumú alkalmazásokat, amelyeket a rendszer megvizsgál, és az eredményeket megjeleníti az érdeklődő számára. A szerver oldalra megérkezett gyanús fájlokon először néhány statikus elemzési lépést hajtok végre. Ez szükséges előkészítés a későbbi vizsgálatok elvégzéséhez. Ezután létrehozok egy friss, emulált környezetet, ami biztosan nem tartalmaz semmilyen károsodást egy korábbi vizsgálat eredményeként. A környezetben elindítok különböző alkalmazásokat, és többféle felhasználói műveletet is szimulálok, hogy az egy valós eszköz állapotához közeli helyzetbe kerüljön. Megfelelő idő eltelte után telepítem a vizsgálandó alkalmazást, amelynek szintén felhasználói utasításokat küldök, ezzel is egy valós helyzetet szimulálva. A rendszert ebben az állapotban szintén tovább hagyom futni, hogy az alkalmazásnak legyen kellő ideje végrehajtani műveleteket. A várakozás után egy kernel modul segítségével elkészítek egy máso-

latot az emulált környezet memóriájának a tartalmáról. Az emulátort ekkor le is állítom, mivel a további vizsgálatok már a memória képen futnak majd. A memória kép analízist a széles körben alkalmazott Volatility keretrendszer segítségével végzem, amit felhasználva különböző értékes információkhoz jutok a rendszer állapotával kapcsolatban. Így hozzá tudok jutni a vizsgált alkalmazás által végrehajtott változtatások listájához, illetve részletesebb képet kapok az alkalmazás viselkedéséről.

Abstract

There are several known technics used by the experts in forensics security investigations and forensics research projects. One of the most common used technic is analysis of volatile memory (live memory forensics). The rise of new technologies requires development and adaptation of investigation methodology. Recently we have just witnessed it in the area of mobile platforms. My thesis focuses on tools supporting memory analyses available on Android platform. I have developed an automated system by developing further and merging currently available solutions. The aim of the software is to perform the tasks in advance that help and support the analysis performed by a human expert. The analysis is performed in an emulated environment designed specifically for this purpose and further analysis. Here it is possible to do secure investigations on potentially malicious software. The system is working as a web service which can be connected at two points. The system has a client application running on android platform. It is investigating further applications installed on the same system. In case an application is suspected potentially harmful than the application installer is sent to the server for further analysis. Another point of access of service is a conventional website. Anyone can upload applications here in ".apk" format which will be examined by the system and the results will be presented for the inquirer. First I perform various static analyses steps on the suspicious files that arrived on the server side. This is necessary before further analyses. Then I create a fresh, emulated environment which surely does not contain any damage caused by previous analysis. I start several applications in the environment and simulate various user inputs in order to put the system in a close to real state. After certain period of time I install the application to be investigated and also send user inputs to this specific application again to simulate a real scenario. I let the system run in this state so that there is sufficient time for the application to perform various operations. After the pause I capture a copy of memory content of the emulated environment with the help of a kernel module. At this point I shut the emulator down because further analysis will only run on memory image. I perform memory image analysis with the help of the well-known Volatility framework. I gain valuable information about the system's state by using

this framework. This enables me to gain access to the list of modifications performed by the application under investigation and get more detailed information about the behavior of the application.

Bevezető

A mobiltelefon-gyártás mára húzóágazatává vált az összes nagy tech-cégnek. Beszédes adat, hogy az Apple 2014 Q4-es jelentésében az szerepel, hogy a cég 8,5 milliárd dolláros negyedéves nyereségének 56%-a az iPhone eladásokból származik. Ehhez még hozzávehető a táblagépek piaca, ami 13%-át képezte a nyereségnek. Összességében így elmondhatjuk, hogy az Apple összbevételének több mint 2/3-a a mobil eszközeiből származik. Ez az arany sok helyen még drasztikusabb.

A fejlett világban mára szinte minden embernél található egy vagy több készülék, amelyek teljesítményüket nézve inkább számítógépnek számítanak már mint hagyományos értelemben mobiltelefonnak. Az iOS és az Android operációs rendszerek megjelenésükkor teljesen felborították a piacot. Az új okos eszközök funkciói messze túlszárnyalják elődeiket. A sikerük főként a hardverek gyors fejlődésének és az újonnan fejlesztett operációs rendszerek gazdag szolgáltatásainak köszönhető.

A mobil operációs rendszerek versenyében jelenleg az Android áll nyerésre. A korábbi versenyzők, például a Nokia Symbian rendszere, szinte már teljesen eltűnt a piacról. A 2013-as kimutatások szerint az Android operációs rendszer rendelkezik a legnagyobb 79,3%-os piaci részesedéssel. Az őt követő gyártó az Apple, melynek piaci részesedése 13,2%. A Microsoft 3,7% és a BlackBerry 2,9%-os piaci részesedése már elenyésző. A fenti számok azt jelentik, hogy minden tíz eladott telefonból legalább hét eszközön, Android operációs rendszer fut. A platform népszerűsége elsősorban nyíltságának és azoknak a gyártóknak köszönhető, akik partneri kapcsolatban állnak a Google-lel és a telefonjaikat ezzel az operációs rendszerrel szállítják. A gyártók legtöbbje tagja a Google által 2007-ben megalapított Open Handset Alliance nevű csoportnak is, amely hardver, szoftver illetve telekommunikációs cégek tömörülése. Az együttműködés célja fejlett nyílt szabványok fejlesztése elsősorban a mobil platformok számára.[10]

Az Android platform nyíltsága és elterjedtsége miatt lett a kártékony kódokat készítő csoportok célpontja. A bűnözők célja nem csak a pénzszerzés és az adatok ellopása, hanem kormányzati kémkedés is lehet. Egy okostelefon, amelyben ennyi szenzor megtalálható, a megfelelő hozzáféréssel igen veszélyes megfigyelő eszköz lehet a támadók kezében. A Kaspersky 2013-as egész éves kimutatásában[14] található

információk szerint, az év elejéhez képest, az év végére megduplázódott az ismert mobil malware minták száma. Ez a növekedés 2011-ben kezdődött és robbanásszerűen nőtt idáig. Feltételezhető, hogy a jövőben is folytatódni fog ez a tendencia. Az ismert malwarek 98%-a az Android platformot célozza meg, ami nem meglepő, mert a kiber bűnözők minden igényét kielégíti: széles körben elterjedt, könnyű fejleszteni rá, és az emberek könnyedén telepíthetik az alkalmazást a telefonra nemcsak a Google Play Store-ból, hanem más alkalmazás boltból és egyéb weboldalakról is.

Nem csak a kártékony programok száma növekszik, hanem ezeknek a programoknak a komplexitása is. Kezdetekben a legtöbb malware funkcionalitása kimerült abban, hogy SMS-t küldött fizetős szolgáltatásokra, agresszívan reklámozott vagy felhasználói adatokat lopott. Ma már képesek rejtőzködni az eszközökön illetve akár újabb malwarek telepítését is megkísérelik. Az operációs rendszer sérülékenységeit kihasználva újabb jogosultságokat tudnak szerezni maguknak, melyekkel át tudják venni az irányítást az eszköz felett úgy, hogy a tulajdonosa nem is látja, hogy a program a háttérben tevékenykedik.

Tudományos Diák Konferencia dolgozatom témájául Android malware-ek detektálását választottam. A malware-ek drasztikus növekedése megkívánja, hogy a védekezési oldalon is folyamatos legyen a fejlődés. Munkám során egy komplex rendszert terveztem meg és készítettem el. A rendszer részben már korábban is ismert és bizonyított technikákat használ fel, részben pedig új módszereket, melyeket én dolgoztam ki.

Munkám során két szoftvert valósítottam meg, ezek az UkatemiSHIELD valamint a APKAnalyser alkalmazások. Az UkatemiSHIELD egy Android alkalmazás, amely betartja a Google által javasolt fejlesztési konvenciókat, és az ilyen keretek közt rendelkezésre álló információk alapján nyújt védelmet a felhasználójának. Ennek a megoldásnak az előnye, hogy széles körben terjeszthető, a használatához nincs szükség az eszköz módosítására. Az APKAnalyser pedig egy analízist végző keretrendszer, amely a vizsgált alkalmazást futtatja, és közben a viselkedéséről gyűjt információkat.

A dolgozatomat a következő részekre osztottam:

- *Bevezetés:* a téma általános felvezetése után áttekintés a kutatáshoz releváns korábbi eredményekről.
- *Rendszerterv:* az elkészült rendszer funkcióinak és szolgálatatásainak áttekintése.
- *UkatemiSHIELD:* az UkatemiSHIELD alkalmazás részletes bemutatása.
- *APKAnalyser:* az APKAnalyser alkalmazás részletes bemutatása.

- *Eredmények:* az elkészült rendszerrel végzett mérések eredményei.
- *Továbbfejlesztési lehetőségek:* az elkészült rendszer erősségei és gyengeségei, továbbfejlesztési lehetőségek.

1. fejezet

Irodalomkutatás

Ebben a részben egy már létező jogosultság alapú malware detekciós környezetet illetve egy rootkit detektáló kutatást mutatok be, amelyek az én munkám kapcsán fontosnak bizonyultak. A hagyományos antivírus szoftverek általában szignatúrák és viselkedési minták alapján próbálnak meg malwareket detektálni. A szignatúra alapján történő detekció a már ismert kártékony kódokra nagyon jól működik. Egy, a fájlban található részletre vagy byte sorozatra keresnek, amit szignatúrának neveznek. A viselkedési minták elemzésénél arra törekednek, hogy a még ismeretlen malware-ek ellen nyújtson védelmet az antivirus szoftver. Gyanús viselkedésnek számíthat például egy futtatható fájl írása a merevlemezre.

Bemutatok még két eszközt, amelyek szintén tudományos kutatás eredményként keletkeztek, és a munkám során nélkülözhetetlennek bizonyultak.

1.1. PUMA

Jogosultság alapú malware detekciós környezet Androidra

Ez a korábbi munka gépi tanulás segítségével próbálja az alkalmazásokat klasszifikálni és általánosan arra használni, hogy malware-eket detektáljon. Ebben a munkában is hivatkoznak a már ismertetett Crowdroidra, amelyben viselkedés alapján próbálták dinamikusan detektálni a malware-eket. A készítők itt más módszereket kerestek és arra jutottak, hogy gépi tanulás technikákat alkalmazva próbálják meg egy alkalmazás jogosultságai alapján eldönteni, hogy az kártékony-e vagy sem. Két adathalmazzal dolgoznak: az egyik a jóindulatú alkalmazások, a másik pedig a malware-ek. A jóindulatú alkalmazásokból 1811 darabot gyűjtöttek össze, amiből a tanításhoz véletlenszerűen kiválasztottak 357 darabot. Ezeket a különböző alkalmazás kategóriákból választották ki. A rosszindulatú minták adathalmazát a VirusTotalról szerezték be. Összesen 4300 mintát szereztek, amiből a duplikációk eltávolítása után ennek csak a töredékét, 249 darabot használták fel. Összesen a

tanításra használt adathalmaz kicsivel több, mint 600 mintát használt fel[15].

A jogosultságok kinyeréséhez szétbontották az Android alkalmazásokat, amik *.apk kiterjesztésű csomagolt fájlok. Ebben a csomagolt állományban található egy AndroidManifest.xml fájl, ami tartalmazza a program által használt összes jogosultságot.

```
<manifest>
<uses-permission />
<permission />
<permission-tree />
<uses-sdk />
....
</manifest>
```

A jogosultságok kinyerése után feature vektorokat hoztak létre, amiket a WEKA nevű szoftverrel elemeztek. A WEKA egy gépi tanulást segítő szoftver, amiben megtalálható előre implementálva minden szükséges eszköz, hogy tudjunk klaszterezni, információ nyereséget számolni, vagy klasszifikálni egy betanított halmaz alapján. A készítők több algoritmust is kipróbáltak és a J48, NaiveBayes, RandomForest és ezek közül a RandomForest bizonyult a legjobb algoritmusnak. Sajnos az eszközök erőforrásai nagyon limitáltak egy ilyen vizsgálat végrehajtására.

1.2. Rejtőzködő Android rootkit detektálás

Robert C Brodbeck munkája a rejtőzködő rootkit-ek detektálásáról szól. Több rejtőzködő rootkit-et valósított meg, amik egy fájlt, folyamatot, modult vagy egy portot rejtettek el. Amikor egy felhasználó a parancssorból megpróbálja kilistázni ezeket, akkor ezek a rejtett információk nem jelennek meg. Kétféle eszközön próbálta ki az eljárásait: emulátoron és egy fizikai eszközön, ami egy Samsung Galaxy Nexus volt. A detektálási módszerek közül többet próbált ki: próbálkozás alapú, integritás ellenőrzés, szignatúra, heurisztika, és viselkedés. Készített több kártékony kódot tartalmazó programot, amikkel a teszteket végezte el. Fontos megjegyezni, hogy a telefonon egy módosított kernel változat futott, amiben a kernel modulok betöltése engedélyezve volt, és saját kernel modulokat használtak. A legeredményesebb megoldás az integritás ellenőrzés volt, természetesen ennek a működéséhez szükséges, hogy megbízható forrásból származzanak a tiszta mintáink, amivel össze lehet hasonlítani az eszközt. A második legeredményesebb megoldás a heurisztikus vizsgálat volt, a többi megoldás sajnos nagyon gyenge eredményeket mutatott[3].

1.3. Volatility

2007-ben a BlackHat DC konferencián AAron Walters és Nick L. Petroni, Jr. bemutatott egy Volatools nevű kutatást[20]. Munkájuk célja az volt, hogy a forensics kutatásokba egy új irányt hozzanak létre: a tartalomvesztő memóriák vizsgálatát. Ennek segítségével a vizsgált eszközök RAM-jának a tartalmát tudták vizsgálni.

A projekt azóta nagy népszerűségnek örvend, idővel megjelent támogatás platformok egész sora számára az azóta már új nevet kapott Volatility-ben. A szoftver legutolsó fő verziójának az egyik jelentős újítása az Android platform támogatása volt. Munkám során kihasználtam ezt az új lehetőséget, az elkészült memóriaképet a Volatility 2.4-es verziójával elemeztem. A felhasznált funkciókat a 4.4 részben mutatom be részletesen.

1.4. LiME

A LiME programot 2012-ben mutatta be Joe Sylva a ShmooCon konferencián[1]. Az eszköz egy LKM (Loadable Kernel Module), amely a linux kernelbe betöltődve képes az eszköz memóriájának a tartalmát kiolvasni. A tartalmat ezután igény szerint vagy elmenti egy adott célterületre a merevlemezen vagy egy hálózati kapcsolaton keresztül elküldi.

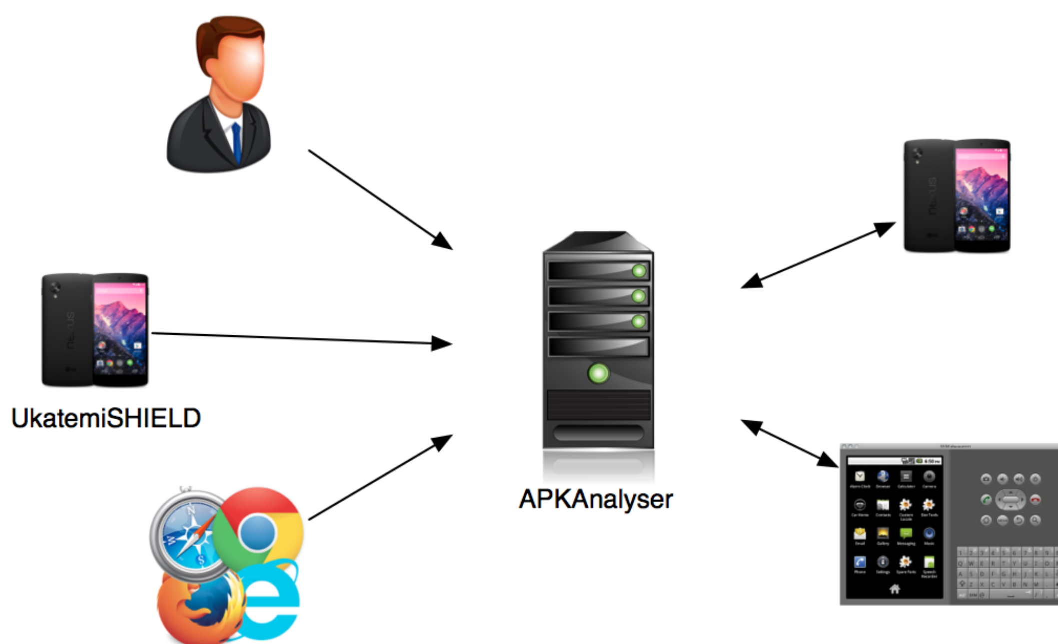
A kutatás során bemutatták, hogy az elérhető egyéb megoldások egy Android alapú eszköz esetében milyen hatékonysággal képesek a RAM tartalmát helyreállítani. Megmutatták, hogy az egyéb megoldások, amelyeket a hagyományos PC-s környezetben használnak a kutatók és az elemzők nem működőképesek a mobil környezetben. Amelyek technikailag mégis működnének, azok pedig működésük során olyan szinten módosítják a rendszert, ami kártékony lehet a vizsgálat szempontjából. Ezek a problémák megállapítása és vizsgálata után fejlesztették ki a LiME szoftvert, amely képes a rendszer legminimálisabb módosítása mellett a memória tartalomhoz a legnagyobb megbízhatósággal hozzáférni.

Ezt az eszközt használtam én is a memóriatartalmak rögzítésére, amit részletesen a 4.3 részben mutatok be.

2. fejezet

Rendszer felépítése

Az elkészült teljes rendszer két fő komponensből áll. Ezek képesek önállóan is feladatot ellátni, de szükség esetén együtt is működnek egy komplex védelem kialakításában. A fő komponensek az UkatemiSHIELD és az APKAnalyser alkalmazások. A rendszer teljes felépítését a 2.1. ábra mutatja be.



2.1. ábra. *Rendszervázlat*

2.1. UkatemiSHIELD

Az UkatemiSHIELD alkalmazás egy védelmi és elemző célokat szolgáló alkalmazás. Folyamatos futtatásakor megadott időközönként ellenőrzi a rendszert, vizsgálja a feltelepített alkalmazásokat, és a fájlrendszeren elérhető fájlokat. Gyanús fájl, vagy

alkalmazás esetén figyelmezteti az eszköz tulajdonosát, hogy a készüléke fertőzött lehet, ezért további lépések megtétele lehet szükséges. Az alkalmazást részletesen a 3. fejezetben mutatom be.

2.2. APKAnalyser

Az APKAnalyser egy Android alkalmazások vizsgálatát végző szolgáltatás. Több felületen keresztül is elérhető, hogy a felhasználók széles köre számára könnyen használható legyen.

A szoftver több szempontból is megvizsgálja a feltöltött alkalmazásokat. Először a VirusTotal adatbázisában keres, hogy a minta nem egy ismert malware-e esetleg. Ezután dinamikus analízist végez, amely keretében rögzíti a rendszerben történt módosításokat, és vizsgálja az alkalmazás által igénybe vett rendszer szolgáltatásokat. Az APKAnalyser működését részletesen a 4. fejezetben mutatom be.

3. fejezet

UkatemiSHIELD [1]

Az UkatemiSHIELD[18] alkalmazás egy széles körben használható Android alkalmazás, amely folyamatosan ellenőrzi annak az eszköznek az állapotát, amelyre feltelepítjük. Az alkalmazás tervezése során fontos szempont volt, hogy csak az általánosan engedélyezett szolgáltatásokat használjuk fel az android rendszerben, annak bármilyen módosítása nélkül. Ezáltal olyan védelmi megoldást tudunk készíteni, ami a felhasználók egy széles köre számára rendelkezésre áll.

3.1. Specifikáció

Az elkészült szoftvernek kommunikálnia kell az Ukatemi különböző szolgáltatásaival. A szoftver futásának automatizáltnak kell lennie és folyamatos védelmet kell nyújtania az eszközön. A felhasználó irányítása nélkül kell ütemeznie a vizsgálatokat és az adatok megosztását az Ukatemi tudásbázisával. A telepített alkalmazásokat és a fájlokat kétféleképpen kell vizsgálnia a szoftvernek.

Az első vizsgálat az alkalmazások jogosultságán alapuló detekciós módszer. Ehhez információkat kell gyűjteni az eszközre telepített összes csomagról. Az összegyűjtött adatokból ezután egy feature vektor generálható, ami bemenetként szolgál egy gépi tanulással betanított rendszernek.

A második vizsgálat a fájlrendszeren megtalálható fájlok integritásának az ellenőrzése. Ezt úgy sikerül elérni, hogy az alkalmazás minden fájlról egy hash lenyomatot készít, amit összehasonlít egy adatbázissal, ahol a már korábban ismert, biztonságosnak tekintett fájlok hash-ei találhatóak.

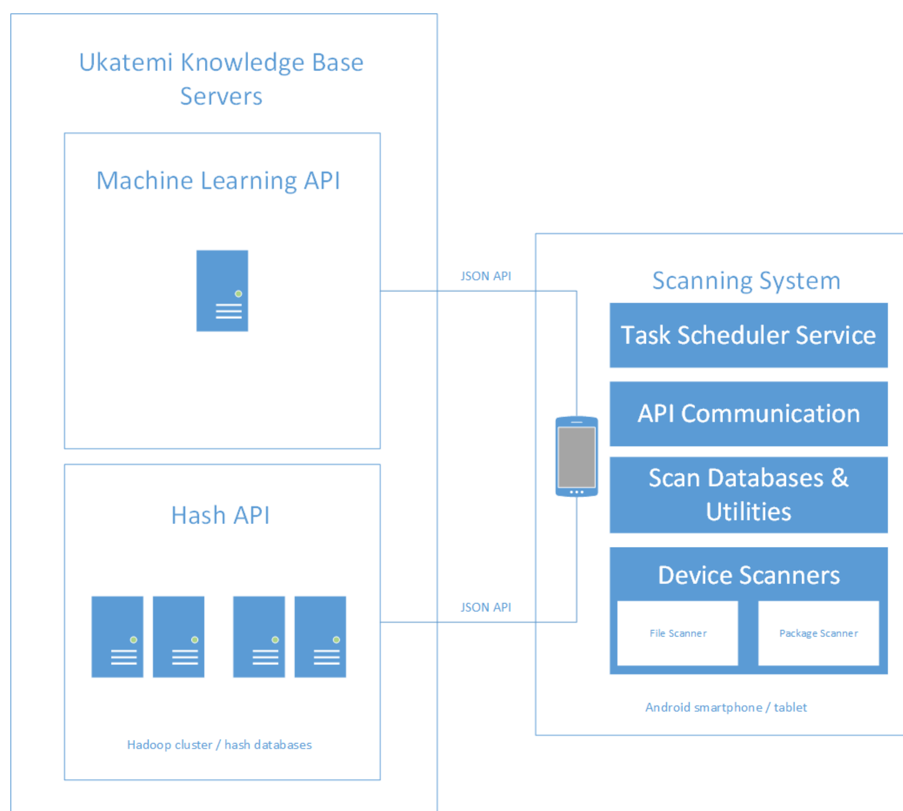
A vizsgálatok eredményeit az alkalmazásnak perzisztensen kell tárolnia és ezt folyamatosan szinkronizálnia is kell a tudásbázissal. A vizsgálatok kiértékeléséhez szükséges szolgáltatások is futnak szerver oldalon, így a vizsgálatok elvégzéséhez ezekkel is szükséges a kommunikáció. A kapott eredményeket az alkalmazásnak értel-

[1] Az UkatemiSHIELD alkalmazás fejlesztése közös munkám Váradi Szabolccsal.

meznie kell tudnia, és amennyiben kártékony alkalmazást vagy fájlokat detektál, azt jeleznie kell az eszköz tulajdonosának, valamint továbbítania kell az APKAnalyser felé további vizsgálatok elvégzésére.

3.2. A rendszer tervezése

Az alkalmazás tervezése során fontos szempont volt, hogy könnyedén bővíthető legyen később. Egy olyan keretrendszerhez hasonló megoldás elkészítése volt a cél, amihez egyszerűen lehet később újabb modulokat fejleszteni. Ez a későbbi továbbfejlesztés, illetve az Android platform dinamikus változásai miatt is fontos volt. Emellett időnként szükség lehet a vizsgálatok lecserélésére vagy átalakítására. Az elkészült rendszer fő komponenseinek a vázlatát a 3.1. ábra mutatja.



3.1. ábra. *Alkalmazás rendszervázlat*

A keretrendszernek két fontos dologról kell gondoskodni. Az egyik a feladatok ütemezése és futtatása. Ez az automatizálás miatt szükséges, így a felhasználónak nem kell aktívan közreműködnie, a program magától tevékenykedik a háttérben, külső beavatkozás nélkül.

A másik fontos feladat a különböző szolgáltatásokkal való kommunikáció megvalósítása. Több szerver oldali szolgáltatásra is szükség van a kívánt működés megva-

lósításához. A szolgáltatások használatával minimalizálhatjuk az alkalmazás kliens oldali terhelését, így az kevésbé terheli az amúgy is erőteljesen limitált akkumulátor kapacitásokat.

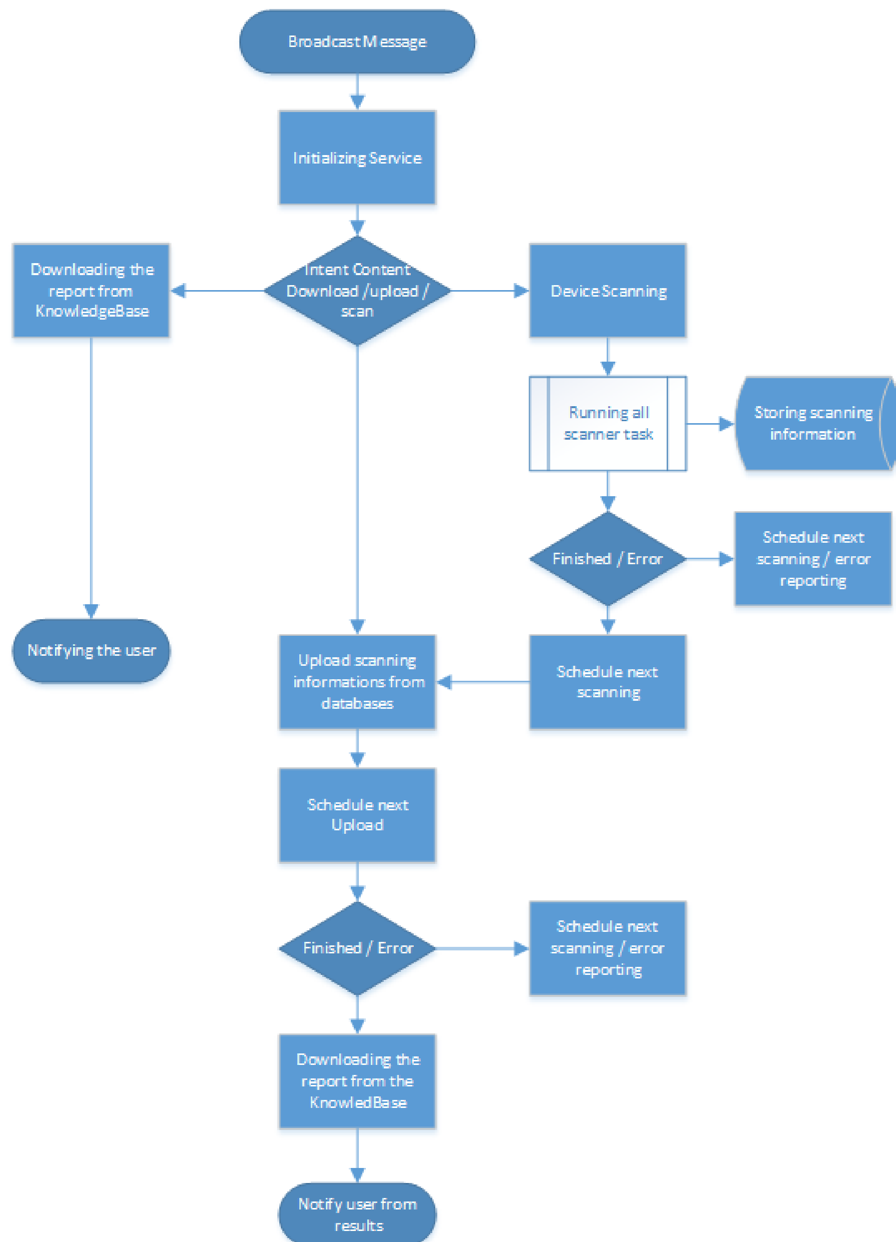
A jogosultság alapú klasszifikáció szerver oldalon fut, mivel annak erőforrásigényét a legtöbb mobil eszköz nem tudná kiszolgálni. A szolgáltatáshoz szükséges volt létrehozni egy tanuló halmazt, ami a tiszta és fertőzött alkalmazások által használt jogosultságból áll. Ehhez mintákat kellett gyűjteni, amiket a VirusTotal szolgáltatásról és a Google Play Store-ból sikerült beszerezni. A tanító halmazban a két csoport aránya nagyjából egyenlő kell legyen, hogy a tanítás kiegyensúlyozott lehessen.

A másik szerver oldalon futó szolgáltatás az a Hash API, ami az ellenőrizni kívánt hash-ekről tudja eldönteni, hogy azok egy már ismert, biztonságosnak nyilvánított file-hoz tartoznak-e. A szolgáltatás mögött egy több gépből álló, elosztott adatbázis szerver található, egy Hadoop klaszter. Az adatok nagy mennyisége és a sok összehasonlítás nagy költsége miatt ennek a feladatnak a megoldása is szerver oldalra került.

Az alkalmazásnak nagyon kis részét teszi ki a felhasználói felület. A program tevékenységének nagy része a háttérben történik. Az egész folyamat automatizáltan fut és figyelembe veszi az eszköz paramétereit is a futás közben. Ha az akkumulátor töltöttségi szintje nem elegendő, akkor az alkalmazás addig fog várni, amíg fel nem töltik a telefont egy bizonyos szintre. Ezalatt semmilyen vizsgálat nem fog futni az eszközön és a hálózati kommunikáció is leáll. Az alkalmazás előre beállított időközönként vizsgálja az eszközt és minden egyes vizsgálat végén kommunikál a tudásbázissal. Amennyiben a vizsgálat vagy a kommunikáció sikertelen, azt a szoftver kezeli és egy későbbi időpontra ütemezi a sikertelen feladatok megismétlését.

3.3. A rendszer működése

Az alkalmazás működését bemutató teljes folyamatábra a 3.2. ábrán látható. A futás egy broadcast üzenet elkapásával kezdődik. Ennek hatására a háttérben futó szolgáltatás automatikusan elindul és elkezdi futtatni a vizsgáló folyamatokat. A folyamatok végeztével a futás nem áll meg, hanem elkezdődik az adatok feltöltése. Ha ez sikeresen befejeződik, akkor az ütemező megállapítja a következő futás időpontját és egy pending-Intentet küld a rendszernek. Ez egy késleltetett intent, amelyben megadható, hogy a rendszer mikor dobja el a broadcast-ot. Amennyiben a futás során valami hiba történt, újraütemezzük a futást egy későbbi időpontra.



3.2. ábra. *Folyamatábra a szoftver futásáról*

Az alkalmazást a fejlesztés során módosítani kellett a tervhez képest, mivel a fejlesztés során bejelentett új Android operációs rendszerben megváltozott a háttérben futó alkalmazások ütemezése, így szükséges volt néhány, az új irányelveknek megfelelő változtatást végrehajtani. A fő változtatás azt jelentette, hogy a rendszernek ezután bármikor joga volt egy háttérben futó alkalmazás leállítására, amennyiben az eszközön erőforrás hiány lép fel. Egy leállítás után az alkalmazás csak felhasználói interakcióra indulhat újra. A szolgáltatások megváltoztatása időzített szolgáltatássá megoldja azonban ezt a problémát, mivel így az alkalmazásnak joga van kikényszeríteni az eszközön a wake_lock zárat, amelynek segítségével felébresztheti

a processzort és a telefon többi perifériáját szükség esetén. Az új IntentService alapú megközelítés összességében erőforrás kímélőbb is mint az eredeti, mivel így nem kell az alkalmazást folyamatosan a háttérben tartani, hanem le lehet állítani azokra az időszakokra, amikor nem végez feladatot. Az időzített broadcast üzenet majd úgyis elindítja/felébreszti az alkalmazást. Az új működés megvalósításához így viszont WakefulBroadcastReceivere-ek használatára volt szükség.

3.4. Az alkalmazás komponensei

Az elkészült alkalmazás teljes szoftver architektúrája a függelék F.1 pontjában található. A főbb komponensek feladatát a következőekben mutatom be.

3.4.1. ShieldService

A ShieldService a háttérben futó fő szolgáltatás, amelynek az ősosztálya az IntentService osztály. Az IntentService egy olyan szolgáltatás, amely nem fut folyamatosan a háttérben, csak amikor egy Intentet kap. Ekkor meghívódik a szolgáltatás onHandleIntent(Intent intent) függvénye, amelyben kezelni lehet a kapott intent-et. A hagyományos Service és az IntentService között több különbség is van. Egy Service-nél gondoskodni kell arról, hogy az eszköz áthelyezze magát alvó üzemmódba. Az újabb rendszereken ilyenkor nem futnak a szolgáltatások, vagy ha szükség van rá, hogy fussanak, akkor a PowerManager segítségével wake_lockot kell létrehozni. A wake_lock ébren tartja az eszköz processzorát és így lehetőség van különböző műveleteket végezni. Ilyenkor vigyázni kell arra, hogy elengedjük a zárat, különben a telefon rövid időn belül lemerülhet. Az IntentService ezzel szemben egy esemény hatására kezd csak el futni. Ha WakefulBroadcastReceivert használunk, akkor az automatikusan kér egy zárat a rendszertől. Az onReceive() függvényben végig jogunk van arra, hogy számításokat végezzünk, így ha elindítjuk egy IntentServicet, akkor az is örökölni fogja a megszerzett wake_lock-ot mindaddig, amíg az onHandleIntent() függvény le nem fut.

Ez az osztály inicializálja a főbb komponenseket a futásakor, melyek: ScanManager, DeviceInformationListener, ShieldKnowledgeBase, és ShieldTimer.

3.4.2. ShieldTimer

A ShieldTimer osztály felelőssége, hogy ütemezze a vizsgálatokat és az információk feltöltését a tudásbázisba. Az ütemezést egy AlarmManager segítségével végzi, amely Brodcast üzeneteket küld a többi komponens számára a megfelelő időpontban. Az osztálynak két fontos függvénye van: az egyik a scheduleNextScan(long interval), a

másik pedig a scheduleNextUpload(long interval). Az elsővel a következő vizsgálat idejét, a másikkal pedig az adatok feltöltésének az idejét tudjuk beállítani.

3.4.3. ShieldKnowledgeBase

Ez az osztály felelős a kommunikációért a tudásbázissal. A vizsgálatok végeztével elindul a feltöltés. Az adatbázisban tárolt adatok közül azokat tölti fel, amelyek meg vannak jelölve feltöltésre. A feltöltés után az adatok várakozó állapotba kerülnek, amíg válasz nem érkezik a tudásbázistól. Megfelelő beállítások mellett az alkalmazás fájlokat is tölthet fel a tudásbázisba.

3.4.4. DeviceInformationListener

Az alkalmazás futása során ellenőrzi az eszköz töltöttségi szintjét és az adatkapcsolat állapotát. Az adatkapcsolatot tekintve három különböző állapot állhat fent: az eszköz a Wi-Fi interfészen kapcsolódik az internetre, mobil interneten, vagy egyáltalán nem kapcsolódik semmilyen formában az internetre. Minden egyes vizsgálat előtt lekérdezzük az akkumulátor töltöttségi szintjét is. Ha ez az érték nagyon alacsony, az alkalmazás nem fogja elvégezni a vizsgálatokat, hanem átütemezi azokat egy későbbi időpontra.

3.4.5. ScanManager

A ScanManager osztály felelős a vizsgáló osztályok inicializálásáért és a vizsgálatok futtatásáért. A vizsgáló osztályok folyamatos információkat közölnek ezzel az osztállyal. Ilyen információ lehet ha a futás véget ért, vagy valamilyen kivétel keletkezik futás közben. Ezeket az állapotokat az osztály megfelelően lekezeli, vagyis vagy újraütemezést végez, vagy folytatja a futást egy következő állapotban.

3.4.6. FileScanner

A fájlok integritás ellenőrzése a feladata. A fájlrendszert bejárva hash-eket készít az eszközön található fájlokról. Jelenleg három különböző hash algoritmust használunk fel, melyek a következőek: md5, sha1 és sha256. Ezeket a hash-eket eltároljuk az eszközön található adatbázisba, majd pedig feltöltjük a tudásbázisba, ahonnan információkat kaphatunk, hogy az adott hash egy ismert fájlhoz tartozik-e. Ezzel a modullal a fájlrendszer állapotát is tudjuk vizsgálni. Az új fájlokról így kiderül, hogy potenciálisan veszélyesek-e vagy sem.

Ha egy rendszer fájl módosul, akkor rögtön látni fogjuk, hogy valami nincs rendben az eszközzel. A rendszer partícióhoz nem férhet hozzá egyik alkalmazás sem. Ha ez

mégis megtörténik, az azt jelenti, hogy valamelyik program root jogosultságokat tudott szerezni, és képes volt újra felcsatolni a fájlrendszert írható állapotban is.

3.4.7. PackageScanner

A telepített alkalmazások vizsgálata a feladata. Az alkalmazásoknak a jogosultságait vizsgálja és egy feature vektort állít elő arról, hogy egy-egy alkalmazás milyen erőforrásokhoz kér hozzáférést az eszközön. Ezt a feature vektort elküldjük a szerver oldali szolgáltatásnak, amely ez alapján dönt az alkalmazás veszélyességéről. A háttérben egy gépi tanuláson keresztül tanított rendszert használunk, amely a betanítását követően dönt arról, hogy az adott alkalmazás malware-e vagy legitim. Ez a folyamat legitim alkalmazásokat is megjelölhet malware-ként a jogosultságok alapján, így bizonyos mértékben lesznek fals pozitív eredmények is. A meglévő feature vektorokat az osztály letárolja az eszközön található adatbázisban, amelyből később a tudásbázis felé tudja kommunikálni azokat.

3.5. Az alkalmazást kiszolgáló szolgáltatások

3.5.1. Hash API

A hash-ek ellenőrzéséhez az Ukatemi tudásbázisát használtuk fel. Ez egy több számítógépből álló Hadoop klaszter, amelyen tároljuk az összes hash-t azokról a fájlokról, amikkel már korábban találkoztunk. Az API-nak JSON formátumban lehet kéréseket küldeni, aminek a vizsgált fájlok hash-eit kell tartalmaznia. Bizonyos idő múlva a szerver válaszol egy jelentéssel, ami tartalmazza, hogy az adott fájlok megbízhatóak-e vagy sem. A mobil eszköznek szükségtelen minden fájlját végig nézni. Érdemes a rendszerfájlokra koncentrálni, mert a kártékony alkalmazások ezeket szokták általában módosítani. Minden egyes futáskor megvizsgáljuk a rendszerfájlokat, hogyha a fájl nem változott, nincs probléma. Ha bármelyik rendszerfájl megváltozott, vagy újak keletkeztek, akkor azt jelezni kell a felhasználónak, mert kártékony tevékenységre utalnak.

3.5.2. Gépi tanulás API

Az alkalmazásokat az elkért jogosultságok alapján próbáljuk megvizsgálni. A jogosultságokból létrehozunk egy feature vektort, amit elküldünk a gépi tanulás API-nak. Ez az API egy előre betanított halmaz alapján működik. A tanító halmaz megközelítőleg 6500 mintából áll, nagyjából fele-fele arányban tartalmazott legitim alkalmazásokat, ahogy malware-ket is. Az API képes meghozni a döntést arról, hogy mennyire veszélyes az adott alkalmazás. A 20 legjobb jellemző kiválasztásával

el tudjuk dönteni, hogy a telepített csomag milyen kategóriába sorolható. Jelenleg két kategória létezik: legitim és malware.

A tanítás során először létrehoztunk egy *.csv fájlt a betöltendő mintákkal, amelyek a már említett két különböző osztályba tartoznak. A programok, amivel ezeket a mintákat feldolgoztuk az igcalc és a WEKA nevű alkalmazások.[2] Az igcalc arra használható, hogy az információ tartalmát kiszámolja a beadott mintáknak. Ezután a WEKA gépi tanulás algoritmus gyűjteménnyel próbáltuk ki a különböző gépi tanulás algoritmusokat, hogy megnézzük melyik az, amelyik a legjobb eredményt adja.

3.6. Az alkalmazás továbbfejlesztése és gyengeségei

Az alkalmazás moduláris felépítéséből látható, hogy könnyedén ki lehet bővíteni az eszközt vizsgáló további osztályokkal. Minden újabb vizsgálatnak csak egy adott interfészt kell implementálni. Ezután a vizsgálatokat már csak a ScanManager osztályban kell példányosítani. Ekkor a ScanManager automatikusan futtatni fogja az új osztályunkat. Lehetőség van továbbá az Android NDK segítségével natív C++-ban is implementálni az alkalmazás egyes részeit.

Érdemes lehet megvizsgálni, hogy natív kódból a rendszer mely részeihez tudunk hozzáférni. A legnagyobb probléma továbbra is az, hogy nem rendelkezhetünk root jogosultsággal az eszközön. A Google Play store-ba feltölthető olyan alkalmazás is, amely képes kihasználni a root-olt eszközökön rendelkezésre álló további jogosultságokat, ám ez szembe megy minden ajánlással, amit a Google javasol a platformon. Sajnos a platform fragmentációjának köszönhetően nem biztos, hogy az alkalmazás az összes készüléken megfelelően működik.

A rendszer gyenge pontja, hogy az ellenőrzések magas szinten valósulnak meg az operációs rendszer architektúrájában. Amennyiben egy rejtőzködő malware-nek sikerül az architektúrában egy alacsonyabb rétegbe beépülni, és szűrni tudja az alkalmazásunk által használt rendszer API-k kimenetét, akkor lehetséges, hogy el tud rejtőzködni előlünk. Ez úgy valósítható meg például, hogy elrejti a fájlokat, amiket használ vagy telepített az eszközre. Így hiába próbáljuk a fájlrendszert vizsgálni és a fájlok integritását ellenőrizni, ezek a fájlok nem lesznek láthatóak számunkra. A malware futása előtt viszont jó esélyünk van arra, hogy detektáljuk a jogosultságai illetve a fájlok alapján, amik felkerültek az eszközre. Szerencsére a rendszeren egyetlen alkalmazás első futása sem indulhat el automatikusan, így a futtatás előtt lehetőségünk van megvizsgálni a telepített csomagot.

A jogosultság alapú detekciónál használt feature vektort is kibővíthetjük további jellemzőkkel, és megvizsgálhatjuk, hogy a detektálási arány javul-e. A rendszerből, ha nehezen is, de kinyerhető, hogy az alkalmazások használnak natív binárisokat.

Root jogosultság nélkül csak úgy férhetünk hozzá ezekhez a könyvtárakhoz, hogyha tudjuk a pontos csomagnevet, amivel az alkalmazást lehet azonosítani. A root exploitokat a kártékony alkalmazások bináris formában tartalmazzák és JNI hívások segítségével hajtják végre azokat. A feature vektort ez alapján ki lehetne egészíteni egy olyan mezővel is, ami azt tartalmazza, hogy van-e az alkalmazásban natív bináris.

Az alkalmazás másik gyenge pontja a különböző szolgáltatásokkal folytatott kommunikáció. Ha a szolgáltatás egyáltalán nem elérhető, akkor az elkészült rendszer nem nyújt védelmet a felhasználónak. A jövőben érdemes lenne legalább részben a telefonon végrehajtani a műveleteket, melyeket jelenleg a szerverek szolgálnak ki.

Hatékony megoldás lehetne még továbbá a root jogosultságot is kihasználni azokon az eszközökön, ahol ez rendelkezésünkre áll. Ennek a jogosultságnak a kihasználásával nem leszünk a rendszer által megszabott korlátok közé beszorítva, és lehetőségünk nyílik átfogóbb és mélyebb vizsgálatok elvégzésére.

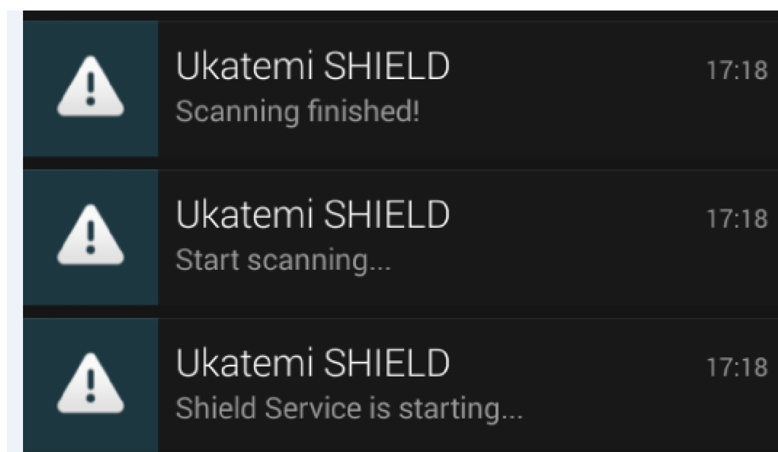
3.7. Az alkalmazás tesztelése

Az elkészült alkalmazást több szempontból is teszteltük. Először megvizsgáltuk, hogy az alkalmazás mint mobil alkalmazás az elvártnak megfelelően működik-e, majd a detekciós mechanizmusokat vizsgáltuk meg.

3.7.1. Alapvető funkciók tesztelése

Az alkalmazás alapvető tesztelése során az ütemező algoritmust, az API kommunikációkat valamint az erőforrás felhasználást teszteltük. A tesztek megnyugtató eredményeket hoztak, az alkalmazás az elvártnak megfelelően viselkedett.

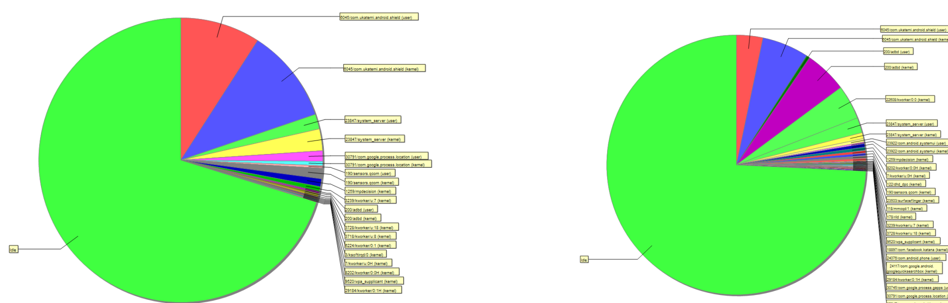
Az ütemező teszteléséhez a megszokottnál kisebb időközöket adtunk meg az egyes műveletek elvégzéséhez, hogy a hibák gyorsabban kiderüljenek. Minden komponens elindulását nyomon lehet követni a log üzenetekben valamint az értesítési központban is. A teszt több napon keresztül futott, hogy minden hibára fény derülhessen. A megjelenített értesítéseket mutatja be a 3.3. ábra.



3.3. ábra. *Az ütemező futásának tesztelése*

Az API kommunikáció vizsgálatakor a működés során előállható összes lehetséges üzenet átvitelét ellenőriztük kliens és szerver oldalon is, és ezeket rendben találtuk.

Az alkalmazás erőforrás felhasználását az Android SDK által nyújtott eszközök segítségével vizsgáltuk meg. Ezzel meg tudtuk állapítani, hogy mennyi memória kerül lefoglalásra a heap-en, valamint, hogy mekkora az alkalmazás CPU terhelése. Ezeket az eredményeket mutatják be a 3.4. valamint a 3.5. ábrák.



3.4. ábra. *CPU terheltség vizsgálat során illetve várakozó állapotban.*

Heap updates will happen after every GC for this client

ID	Heap Size	Allocated	Free	% Used	# Objects
1	18,363 MB	16,849 MB	1,515 MB	91,75%	48 734

Cause GC

3.5. ábra. *A heapen lefoglalt memória mérete.*

A 3.4. ábrán kék (kernel) és piros (user) színnel látható az, hogy az alkalmazás mennyire terheli a processzort. Futás közben a processzoridő 18-20%-át használja az alkalmazás. Amikor az eszközön éppen nem végez műveleteket, csak a háttérben fut,

akkor ez az érték visszaesik 8-10% környékére. A memória használatot is figyeltem, ez átlagosan 18 és 25 Megabyte között mozgott a végzett műveletektől függően.

3.7.2. Hash API tesztelése

A megfelelő teszteléshez elő kellett állítani egy adatbázist a szerver oldal számára a megbízható fájlok hash-eiről. Ehhez a vizsgált eszközt visszaállítottuk teljesen gyári állapotba, majd lefuttattuk rajta az UkatemiSHIELD hash adatbázis generáló modulját. Az így keletkezett lenyomatokat tekintettük ezután a megbízható és legitim fájlok hash-einek.

Ezután kézzel elhelyeztünk néhány általunk ismert gyanús fájlt az eszköz háttértárján, majd vártuk, hogy megérkezzen az értesítés az újonnan megjelent fájlról. A gyanús fájl egy su bináris volt, amit a system partícióra másoltunk fel. Az innen futtatott programok magasabb privilégium szinttel rendelkeznek, így ez az egyik leggyakoribb példa az eszközök root-olására. Az ötlet a DroidDream malware családon alapul, ami szintén ezt a műveletet végezte el. Rövid idő elteltével az elvárásainknak megfelelően a detekció megtörtént, így állíthatjuk, hogy ez a szolgáltatás megfelelően működik.

3.7.3. Gépi tanulás tesztelése

A tanítás során a mérések alapján a RandomForest algoritmust felhasználva működik a legmegbízhatóbban a klasszifikációs algoritmus. Az így mért eredményeink némileg elmaradtak a mások által hasonló mérések során kimutatott eredményektől. Megvizsgáltuk, hogy ennek mi lehet az oka, és a következőre jutottunk. Az általunk használt tanító mintahalmaz mérete jóval nagyobb volt, mint az a halmaz, amit a referenciaként használt PUMA[15] kutatás során használtak, ami nagyban befolyásolta a kapott eredményeket. A halmaz méretének csökkentésével sikerült előállítanunk olyan részhalmazt, amire a klasszifikációs algoritmus találati aránya megemelkedett, így reprodukálva a korábbi eredményeket. Ennek ellenére mi maradtunk az eredeti tanító halmaznál, mivel bár így egy rosszabb találati arányt kapunk bizonyos mintákra, összességében a nagyobb tanító halmaz miatt valószínűleg a kártékony kódok egy szélesebb halmazát sikerült lefedni, ezzel pedig hosszú távon hatékonyabb működést érhetünk el.

4. fejezet

APKAnalyser

Az APKAnalyser szolgáltatás célja egy Android alkalmazás viselkedésének vizsgálata. Ez a szoftver az UkatemiSHIELD (3. fejezet) alkalmazással együtt egy komplex védelmi megoldást képest nyújtani. Emellett önálló szolgáltatásként alkalmazások vizsgálatára önmagában is alkalmas.

4.1. A rendszer leírása

Az APKAnalyser több különböző felületen keresztül elérhető szolgáltatás. Miután a felhasználó megad egy vizsgálandó fájlt, a szoftver egy módosított környezetben futtatja azt, majd a környezet memória tartalmát rögzíti, és azt elemezve képes az alkalmazás viselkedésével kapcsolatban információkat megállapítani. Emellett a minta hash-e alapján keres a VirusTotal weboldalon, hogy megállapítsa, hogy a minta egy ismert malware-e már.

A szolgáltatás célja kettős. Egyrészt az elvégzett elemzések alapján az APKAnalyser eldönti, hogy a vizsgált alkalmazás viselkedése gyanús-e vagy sem. Emellett előkészít minden lehetséges információt egy humán szakértő számára, hogyha később az alkalmazás kézi vizsgálatára kerül a sor, akkor a folyamat a lehető leggyorsabb lehessen.

4.1.1. Hozzáférési felületek

Az APKAnalyser 3 különböző módon érhető el a felhasználói számára. A szoftver alapvetően egy webszolgáltatásként fut, így a legegyszerűbben egy webes felületen keresztül érhető el. Itt lehetősége van a felhasználónak .apk fájlok feltöltésére (egyszerre akár többre is), amelyeket ezután a rendszer különböző elemzéseknek vet alá.

A második mód, ahogy hozzá lehet férni a szolgáltatáshoz az maga az UkatemiSHIELD alkalmazás. Amennyiben az UkatemiSHIELD egy alkalmazást kártékonynak vagy legalább gyanúsnak talál, akkor annak telepítő fájlját elkül-

di az APKAnalyser számára további vizsgálatra. Ehhez a kommunikációhoz az APKAnalyser rendelkezik egy mulipart fájl feltöltő felülettel is, mivel a mobil alkalmazások az eszközök korlátos kapacitásai miatt csak ilyen formában tudnak nagyobb fájlokat elküldeni hálózaton keresztül.

A harmadik lehetőség, a szolgáltatás elérésére, csak azok számára áll rendelkezésre, akik hozzáférnek a szolgáltatást futtató számítógéphez. Ebben az esetben lehetőség van arra is, hogy a szerver egy mappájába másoljuk a fájlokat, majd elindítsuk az APKAnalyser-t, amely feldolgozza a fájlokat a megadott mappából. Ez a megoldás a mérések és a tesztelések során bizonyult hasznosnak, mivel így könnyen és gyorsan lehetőség volt nagy mennyiségű analízis elindítására.

4.1.2. Analízis környezet

Az APKAnalyser-t úgy készítettem el, hogy többféle környezetben is képes legyen vizsgálatot végezni. Ez több szempontból is hasznos. A legkézenfekvőbb előnye ennek a megközelítésnek természetesen az, hogy egy adott fájlt több környezetben is tudunk vizsgálni, így részletesebb képet kaphatunk a viselkedéséről. Emellett más szempontok is felmerülnek még. Lehetőségünk van a vizsgálatokat fizikai eszközön futtatni (az eszköz némi módosítása után), így a potenciális malware egy valós környezetben fut, tehát nagyobb eséllyel nem detektálja az analízis tényét. Ugyanakkor a vizsgálatokat futtathatjuk egy emulátorban is, aminek viszont nagy előnye, hogy tisztán szoftveres megoldás lévén, könnyebben megoldható a skálázhatósága. Az analízis környezet kialakítását részletesen a 4.2. részben mutatom be.

4.2. Analízis környezet kialakítása

Az analízis elvégzéséhez a futtató környezet módosítása szükséges. Lehetőség van emulátor alapú, illetve fizikai eszköz alapú analízis futtatására is, ezért a következőekben bemutatom, hogy ehhez milyen lépések szükségesek. A memória rögzítést végző LiME szoftver egy LKM (Loadable Kernel Module). Ezt a modult futásidőben lehet betölteni a kernelbe, amely a betöltés után rögtön elindul, és végrehajtja a rögzítéshez szükséges lépéseket. Ennek használatához tehát egy kernel modul betöltésére van szükségünk. Erre csak root jogosultsággal van lehetőség, így az analízis során használt eszközt root-olnunk kell.

További nehézséget okoz, hogy az Android operációs rendszerben található kernelben ki van kapcsolva a lehetőség az LKM-ek betöltésére. Ez biztonsági okokból lett így beállítva, mivel így a kártékony kódok nehezebben tudnak a kernelhez hozzáférni. Ezt nem lehet utólag módosítani, a megoldás a problémára az, hogy a kernel forráskódjának beszerzése után módosítani kell a konfigurációs fájlt, és így újrafordítani

a kernelt.

A kernel forráskódja természetesen eltérhet az egyes Android alapú eszközökön, valamint az emulátoron is. Licence okokból kifolyólag a gyártóknak kötelező forráskód szinten elérhetővé tenniük a használt kerneljüket, így ezek beszerzése nem jelenthet gondot.

A lépéseket elvégeztem egy fizikai eszközön is: egy Sony Xperia Arc S[17] készülékhez töltöttem le a kernelt, amelyet a megfelelő módosítások után sikerült is lefordítanom. Emellett az Andorid emulátorhoz is fordítottam új kernelt, hogy emulált környezetben is futtattathassam a vizsgálatokat.

4.2.1. Fizikai eszköz előkészítése

Bootloader unlock - a bootloader kinyitása

Amennyiben új kernelt szeretnénk a telefonra fordítani, ahhoz a telefon bootloader-ét unlock-olni kell. Ez ahhoz szükséges, hogy módosítani lehessen azt a kernelt, amely eredetileg a telefonon található. Szinte minden telefonnál más eljárással kell a bootloader-t unlock-olni. Vagy valamilyen exploit-on keresztül érhető el a kívánt végeredmény, vagy pedig a gyártó maga kínál egy egyszerű megoldást erre. A Sony telefonoknál például létezik erre egy szolgáltatás: megadjuk a telefon IMEI számát, majd megkapjuk e-mailben azt a kódot, aminek a segítségével unlock-olhatjuk a bootloader-t. Ehhez a fastboot parancssoros alkalmazásra van szükség, amelyet a Google, mint univerzális flashtool-t fejleszt az Android alapú eszközökhöz. Ezt a folyamatot mutatom be, ugyanis ez a "szabványos" eljárás. A gyártók azonban szeretnek egyedi megoldásokat alkalmazni mivel ez megengedett számukra.

A Sony telefonok számára a bootloader-t kinyitó kódot a következő oldalon lehet igényelni: `http://unlockbootloader.sonymobile.com`. Az ott leírt lépéseket követve hamarosan rendelkezésre fog állni az a kód, aminek segítségével kinyitható a bootloader. Miután megkaptuk a kódot, szükséges, hogy a telefont fastboot módba tegyük. Ez a lépés szintén erősen gyártó függő, még az sem teljesen egyértelmű, hogy adott eszközön támogatott-e ez egyáltalán. A Sony telefonokhoz itt találjuk a szükséges gombnyomásokat: `http://unlockbootloader.sonymobile.com/fastboot-buttons`. (Az Android piac egyik legnagyobb versenyzője a Samsung előszeretettel alkalmazza a saját megoldását, amit "download" módnak nevez, és ehhez a saját Odin nevű flashtool-ját kell használni, ami egy saját protokollon keresztül kommunikál a telefonnal.) A művelet elvégzéséhez természetesen telepíteni kell a legfrissebb Android SDK-t, továbbá Sony eszközökön szükséges az Sony ADB USB driver-e, amit le lehet tölteni a Sony oldaláról. Miután a telefont fastboot módba tettük, a következő parancsot kiadva tudjuk unlock-olni a bootloader-t:

```
fastboot.exe -i 0x0fce oem unlock 0xKEY
```

ahol a KEY helyére kell azt a kódot beírnunk, amit kaptunk a gyártótól és az -i kapcsolóval pedig a gyártó azonosítóját "VendorID"-t tudjuk megadni.

Kernel fordítás

Miután letöltöttem a kernel forrását, le kellett fordítanom egy cross compiler segítségével, hogy a telefonra tudjam tölteni. Ehhez szükségem volt egy ARM-GCC cross compiler-re, méghozzá arra a verzióra, amivel az eredeti kernelt is fordították. Szükségem volt a fordításhoz még a config fájlra, ami alapján a fordítás történt. Ezt a

```
make *codename*_defconfig
```

paranccsal tudtam létrehozni. A codename helyére a telefon gyártón belüli kódnevét kell írni, például a Sony Ericsson Experia ARC esetében ez semc_anzu_defconfig. Miután meglett a konfigurációs fájl, lefordítottam a kernelt a következő paranccsal:

```
ARCH=arm CROSS\_COMPILE=/opt/arm-2010q1/bin/arm-none-eabi- make.
```

A CROSS_COMPILE részhez a gcc-nek az elérési útját kellett betennem a megfelelő prefixszel együtt. Amikor a kernel fordítása elkészült, akkor az arch/arm/boot/ mappában található zImage fájl a tömörített kernel-image.

Boot image készítés

A boot image készítéséhez három dologra volt szükség, melyek a következők: kernel-image, ramdisk, mkbootimg. A kernel-image a már előbb lefordított kernel, amit zImage formátumban kaptam meg.

A kernel betöltésekor a ramdisk (initrd) meghajtó kerül mount-olásra, majd innen további modulok töltenek be. Ez néhány könyvárat és fontos futtatható állományokat tartalmaz. Ilyen például az insmod bináris, aminek segítségével a kernel képes modulokat betölteni. A ramdisk készítésének legegyszerűbb módja, hogyha egy gyári firmware boot.img-ből kicsomagoljuk és azt használjuk a saját kernelünkkel. Ez a megoldás nagy valószínűséggel az összes gyártónál megegyezik.

A mkbootimg egy olyan eszköz, aminek a segítségével a kernel-image-ből és a ramdisk.img-ből boot.img-et tudunk csinálni. A szoftvert a következő paraméterekkel kell futtatni:

```
mkbootimg --base 0x00200000 --kernel kernel/arch/arm/boot/zImage
          --ramdisk ramdisk.img -o boot.img
```

Boot image feltöltése a telefonra

A gyári fastboot eszköz segítségével ez egy igen egyszerű feladat. A programot a következő paraméterekkel futtatva tölthetjük fel az új boot.img-et egy eszközre:

```
fastboot [-i 0x0fce] boot boot.img
```

ahol az -i kapcsoló a VendorID csak opcionális. Ezután újra kell indítani a telefont, majd miután betöltött az Android, a rendszer beállítások között a telefon verziószámánál ellenőrizhető, hogy az újonnan elkészült kernel töltött-e be induláskor. Amennyiben a telefon nem indul el, a folyamat során valahol hiba történt. Ekkor a telefon egyszerű újraindításával lehetőségünk van ismételten az eredeti kernel-t használni. Ez azért lehetséges, mert a fenti paranccsal az új boot-image csak a memóriába került betöltésre és az operációs rendszer is csak onnan boot-olt be.

4.2.2. Emulátor előkészítése

Az emulátor módosítása során lényegesen egyszerűbb feladatot kellett megoldanom, mint a fizikai eszköz kapcsán. Első lépésként az emulátorhoz is le kellett töltenem a kernel forráskódját, ami elérhető a Google git szerveréről[6]. Miután a kernelt beállítottam ugyanúgy mint korábban, a Google által mellékelt toolchain segítségével sikerült is azt lefordítani. Az elkészült kernellel viszont sokkal könnyebb elindítani az emulátort mint egy fizikai eszközt. Ebben az esetben csak arra van szükség, hogy az emulátor elindításakor egy paraméter segítségével megadjuk az elérési útvonalat a zImage fájlra:

```
./emulator -avd malwareAVD -kernel binaries/kernel/zImage -show-kernel -verbose
          -partition-size 300 -sdcard files/sdcard
```

A további szükséges paraméterek: -avd: az AVD neve, amit futtatni szeretnénk; -show-kernel -verbose: verbose log-olás aktiválása; -partition-size: a rendszer partíció méretének az új értéke; -sdcard: a virtuális SD kártya elérési útvonala, ami megjelenik az emulátorban.

A rendszer partíció mérete úgy van meghatározva, hogy pont akkora legyen mint a Google által rámásolt fájlok mérete, ezzel is gátolva azt, hogy egy támadó kártékony kódot tudjon elhelyezni erre a partícióra. A vizsgálat során azonban szükségem volt az emulátor root-olásához fájlokat erre a partícióra másolni, így a partíció méretet meg kellett változtatnom.

4.2.3. LiME előkészítése

A LiME program egy LKM, így ennek fordítása során szükség van annak kernelnek a forráskódjára, amibe majd be szeretnénk tölteni a modult. Ebből is látszik, hogy a LiME-ot le kellett fordítani egyszer a Sony telefonhoz, egyszer pedig az emulátor kerneljéhez. Mivel az előző lépés miatt már az összes forrásfájl és eszköz rendelkezésre állt, ezért ez nem okozott gondot.

4.2.4. Analízis környezet root-olása

Kernel modulok betöltéséhez root jogosultságra van szükség, így az emulátort és a Sony telefont is meg kellett root-olni. Ehhez az interneten széles körben elterjedt technikákat alkalmaztam.

4.3. Memória tartalom rögzítése

A LiME modul kétféle üzemmódban képes futni. Lehetőség van a memóriaképet az eszköz SD kártyájára rögzíteni, vagy egy hálózati kapcsolaton át elküldeni egy másik számítógép számára. Én az utóbbit választottam, mivel így biztosabban nem ütközök kapacitás korlátokba, valamint az SD kártya tartalma sem módosul, ami a későbbi vizsgálatok szempontjából értékes lehet.

Az emulátor alapú vizsgálatokkor így arra volt szükség, hogy az adb (Android Debug Bridge) segítségével egy port forward-ot hozzak létre, így lehetőségem nyílt az analízist végző gépről becsatlakozni az emulátor egy adott portjára, ahol a LiME várta a kapcsolatot. Az ehhez elvégzett lépések a következőek voltak:

```
./adb push binaries/lime.ko /sdcard/lime.ko
./adb forward tcp:4444 tcp:4444
./adb shell insmod /sdcard/lime.ko "path=tcp:4444 format=lime"
```

Ezekkel a parancsokkal először az emulátorra másoltam az elkészült LiME kernel modult, majd beállítottam a port foward-olást. Végül pedig betöltöttem a kernelbe a LiME modult, ami így automatikusan elindult TCP alapú átviteli módban, a 4444-es porton csatlakozókra várakozva. Az utolsó paraméter amit meg kellett adnom az az elkészült memóriakép formátuma, aminek a lime-ot választottam, mert ezt könnyű volt a későbbiekben értelmezni.

4.4. Memóriakép elemzése

Az elkészült memóriaképet ezután a Volatility[5] keretrendszer segítségével elemeztem. Az Androidról készült memóriaképek Linux alapú operációs rendszer révén elemezhetőek a Linux rendszerek elemzésére elkészült parancsok segítségével. A Volatility futtatása során minden alkalommal meg kell adni az elemezni kívánt fájlt, valamint a formátumát a fájlnak, hogy a Volatility képes legyen értelmezni azt. Az elemzések során a Volaility-t így futtattam:

```
python vol.py --profile=LinuxEvo4Gx86 -f /memory_images/memory.lime [command]
```

A parancs command része volt a vizsgálatok során mindig változó paraméter. A hasznos információkkal szolgálható paramétereket a következőekben szeretném bemutatni.

Ezeknek a parancsoknak a futtatása egy humán szakértő számára sok információt tartalmazhat a rendszer általános állapotáról. Amennyiben például a kernel struktúrákban módosítást detektál a Volatility, akkor feltehetően rootkit-el van fertőzve a rendszer. Az APKAnalyser lefuttatja ezeket a parancsokat előre, hogy megkönnyítse és felgyorsítsa a későbbi analízist. A szükséges elemzések listája egy konfigurációs fájlban adható meg.

4.4.1. Folyamatok vizsgálata

Azokat a Volatility parancsokat tekintem itt át, amelyek információkkal szolgálnak a futó folyamatokról.

- **linux_pslist:** Listázza a futó folyamatokat, és az azokhoz tartozó pid, uid, gid értékeket, valamint a folyamatok indításának az időpontját.
- **linux_pstree:** Listázza a futó folyamatokat aszerint, hogy azok egymáshoz képest hogy helyezkednek el szülő-gyerek viszonylatban. Amennyiben például egy folyamatnak az ssh process a szülője, az gyanús lehet, mivel ez tipikus jele annak, hogy a rendszeren backdoor-t helyeztek el.
- **linux_psaux [-p pid]:** Kimenete megegyezik a Linux rendszereken megszokott "ps aux" paranccsal. Amennyiben a -p kapcsoló is meg van adva, akkor csak az adott folyamatról ír ki részletesebb információkat.
- **linux_proc_maps -p pid:** Listázza a megadott process memória térképét.
- **linux_dump_map -s add:** Az add címtől kezdve kimenti a memória lap tartalmát egy fájlba. Így lehetőség van például egy bináris kimentésére, amely ezután futtatható vagy tovább vizsgálható. Ennek segítségével többek közt az is detektálható, ha egy folyamatba egy másik kódot injektált.

4.4.2. Hálózati forgalom vizsgálata

A hálózati forgalomról is lehetséges információkat szerezni. Megtudhatjuk például, hogy az eszköz milyen hálózatokhoz van csatlakozva, illetve a kimenő bufferek tartalmát is vizsgálhatjuk, ami árulkodó lehet a kommunikációkra nézve is. Emellett a cache-eket megvizsgálva hozzájuthatunk információhoz arra nézve is, hogy az utóbbi időben milyen szerverekkel bonyolított forgalmat az eszköz. Az interface-ek állapotáról a következő parancs adhat információt:

- **linux_ifconfig:** Az elérhető interface-ek listája, azok állapotával együtt. Ebből tudhatjuk meg, hogy az eszköz milyen hálózatokhoz volt csatlakozva.

4.4.3. Fájlrendszerek vizsgálata

A megnyitott fájlokról szintén találhatunk nyomokat a memóriaképben. Ezek részben be is lehetnek töltve a memóriába, így lehetőségünk lehet a tartalom visszaállítására is. Egy másik hatékony eszköz lehet a tempfs fájlrendszerek vizsgálata. Egy ilyen fájlrendszer tisztán a memóriában található, ezért más analízis módszerrel nem is lehet a tartalmukhoz hozzáférni. Az Android a tempfs partíciókat cache-elési célokra hozza létre, hogy felgyorsítsa például a böngészést az eszközökön. Ebből fakadóan a tempfs fájlrendszer lementése és az ott található adatok elemzése hasznos információkat szolgáltathat például az utoljára letöltött URL-ekkel kapcsolatban, mivel ezek tartalma részben itt megtalálható. A tempfs fájlrendszer a következő parancsokkal vizsgálható:

- **linux_linux_tmpfs -L:** Listázza a memóriában található tempfs partíciókat.
- **linux_tmpfs -S id -D dest:** Az id azonosítójú partíció tartalmának a lementése a dest mappába. Így tudjuk a partíció tartalmát vizsgálni.

4.4.4. Rootkit-ek vizsgálata

Volatility segítségével a memóriaképben lehetőségünk van különböző anomáliákat keresni, amelyek rootkit jelenlétére utalhatnak.

- **linux_psxview:** Végignézi a kernelben található összes struktúrát, ahol a futó folyamatoknak szerepelniük kell. Amennyiben nem mindenhol pontosan ugyanazt találja, akkor valószínűleg egy folyamat megpróbál rejtőzködni valamilyen módon. Ez szinte garantáltan kártékony viselkedést jelent.
- **linux_check_fop:** A fájlművelet végzéséért felelős kernel hívásokat ellenőrzi, hogy nem lett-e azok közül valamelyik átírva. Amennyiben egy kártékony alkalamzásnak sikerül egy kernel hívásba hook-ot elhelyezni, akkor lehetősége lehet a kernel hívás eredményét módosítani. Ez jelen esetben azt jelenheti például, hogy a fájl listázó függvény hívás eredményét módosítva, egy fájl így elrejthető a fájlrendszerből.
- **linux_check_afinfo:** Az előző művelethez hasonló ez az ellenőrzés is. Ebben az esetben a nyitott hálózati kapcsolatok elrejtése a célja egy malware-nek, és ezen tevékenység detektálható itt.
- **linux_check_modules:** Betöltött kernel modulok listáját ellenőrzi több forrásból. Amennyiben eltérést fedez fel, az gyanús tevékenységre utal.

4.5. Automatizált detekció

Az APKAnalyser fejlesztése során kidolgoztam egy módszert, aminek segítségével képet kaphatok egy alkalmazás működéséről. A gyűjtött információk alapján a szolgáltatás képes egy értéket rendelni a vizsgált programhoz, ami ha egy meghatározott határérték fölé esik, akkor a vizsgált alkalmazást kártékonynak tekinthetjük.

A módszer alapötlete, hogy a vizsgált alkalmazás memóriaterületének az elemzése során, lehetőség van a kódban található rendszerhívásokat visszaállítani. Ennek segítségével megtudhatjuk, hogy mihez és hogyan próbál hozzáférni, illetve milyen szolgáltatásokat vesz igénybe az adott alkalmazás.

4.5.1. Előkészítés

A nyers analízis környezetbe először telepítem majd elindítom a vizsgált alkalmazást. Ekkor az emulátor még nincs root-olva, és a lime modul sincs betöltve. Egy átlagos emulátorhoz képest csak az egyedi kernel a különbség. Ezután az adb monkey[8] szolgáltatást használva különböző felhasználói eseményeket szimulálok. Ennek eredményeként más alkalmazások is elindulnak, az emulátorra SMS érkezhet, illetve a kijelző több különböző helyen is érintést érzékel. A stimuláció után megpróbálom újra elindítani a vizsgált alkalmazást. Amennyiben az már futott, akkor nem indul el újra az alkalmazás, csak újra előtérbe kerül. Gyakran előfordul azonban, hogy a kártékony alkalmazások nem a legjobb minőségűek, így felhasználói felületük tesztelésével könnyen hibát is tudunk okozni. Ha ez előfordulna, akkor az alkalmazás újbóli elindításával garantálom, hogy az mindenképp fusson. Ekkor egy rövid várakozás után végrehajtom a szükséges módosításokat az emulátoron, majd elindítom a memóriakép rögzítését. Az átlagosan nagyjából 500 MB körüli adatmennyiség átvitele néhány percig is eltarthat. Az átvitel befejeztével az emulátort leállítom, mert innentől nincs rá szükség.

4.5.2. Folyamat vizsgálata

Az emulátort minden vizsgálat előtt újonnan hozom létre, hogy tiszta rendszer álljon rendelkezésre. Így ezután minden módosítás biztosan az adott alkalmazás viselkedésének a következménye. Ebből adódik, hogy a rendszerben futható folyamatok ismertek. A Volatility-nek a linux_psaux parancsát futtatva, a kimenetben az egyetlen ismeretlen sor a vizsgált alkalmazáshoz tartozó folyamatot írja le. Ezt megtalálva könnyedén megállapíthatjuk a folyamat pid-jét (process id), amely szükséges néhány további parancs futtatásához. Egy példa kimenete a vizsgálat ezen lépésének:

```
Pid     Uid     Gid     Arguments
...
750     10037   10037   com.android.providers.calendar
819     10027   10027   com.android.defcontainer
836     10028   10028   com.svox.pico
887     1000    1000    com.android.keychain
914     10047   10047   com.camelgames.app.BatteryMonitor
1043    0       0       /system/bin/sh -c insmod /sdcard/lime.ko "path=tcp:4444 ...
1045    0       0       insmod /sdcard/lime.ko path=tcp:4444 format=lime
...
```

A pid ismeretében lehetőség van a folyamat címterét megvizsgálni (linux_proc_maps opció), így megtudhatjuk, hogy pontosan milyen binárisok hova voltak map-elva a folyamat címterében. Minden alkalmazás futtatása során rengeteg bináris kerül be a címterükbe. Erre azért van szükség, hogy a rendszer egyes részeit, valamint a futáshoz szükséges binárisokat az alkalmazások gyorsan és könnyedén elérhessék. Ennek megfelelően minden alkalmazás címterében megtalálható a futáshoz szükséges keretrendszer (framework.apk), a zygote vagy az init bináris (az Android azon binárisai amelyek a folyamatok elindításáért felelősek), valamint a Dalvik virtuális gép binárisa is. Ezek mellett természetesen a memóriatérkép vizsgálatával megtalálhatjuk, hogy a futtatandó alkalmazás kódja hova került betöltésre a memóriába.

Az Android alkalmazások java kódját a Dalvik virtuális gép futtatja. Ennek megfelelően az alkalmazás futtatható fájljai nem natív kódot tartalmaznak, hanem Dalvik byte kódot. Ebből adódik az az elsőre furcsának tűnő helyzet, hogy az alkalmazás kódja a memóriában egy, csak-olvasható, de nem futtatható lapon található. A kódot ugyanis innen a Dalvik virtuális gép csak kiolvasni fogja, futni pedig már csak a virtuális gép saját kódja fog. A következő példa egy alkalmazás memória térképéből egy részlet:

```
914 0x000000004af7e000 0x000000004af80000 r-- 0x7b000 31      1
       679 /data/app/app.BatteryMonitor-2.apk

914 0x000000004af80000 0x000000004af81000 r-- 0x7c000 31      1
       679 /data/app/app.BatteryMonitor-2.apk

914 0x000000004af81000 0x000000004af92000 r-- 0x0         31      1
       739 /data/dalvik-cache/data@app@app.BatteryMonitor-2.apk@classes.dex
```

Ezután ezeket a memória területeket kimentve fájlokba visszakapjuk az alkalmazás futtatható .dex (dalvik executable) binárisát. Ezeket a fájlokat, amelyek Dalvik byte kódot tartalmaznak, tovább elemezhetjük. A Dalvik bytekódban olvasható formában megtalálhatóak az egyes osztályok és azok metódusai, amelyek a forráskódban szerepelnek. Ugyanitt a kódban szereplő operációs rendszerhívásokat is megtalálhatjuk ezekben a fájlokban. A 4.1. ábra bal oldalán az elemzés végén helyreállított Dalvik byte kód részlet, míg a jobb oldalon a vizsgált alkalmazás APKTool[19] segítségével visszafejtett Dalvik byte kódja látható.

```
:pdus..get..[Ljava/lang/O
:bject;..[Landroid/teleph
:ony/SmsMessage;..Landroi
:d/telephony/SmsMessage;.
:.createFromPdu..getDispl
:ayMessageBody..getDispla
:yOriginatingAddress..pre
:ferences_data..getAll..L
:l;..\|..+..abortBroadcas
:t..content://sms.._id=?.
:.ILLL..delete..Ljava/uti
:l/Random;..II..nextInt..
:Ljava/lang/Thread;..J..V
:J..sleep."Landroid/telep
:hony/gsm/SmsManager;..ge
```

```
539     check-cast v4, [B
540
541     check-cast v4, [B
542
543     invoke-static {v4}, Landroid/telephony/SmsMessage;->createFromPdu([B)
        Landroid/telephony/SmsMessage;
544
545     move-result-object v4
546
547     aput-object v4, v7, v8
548
549     add-int/lit8 v4, v8, 0x1
550
551     move v8, v4
```

4.1. ábra. *A helyreállított és az eredeti Dalvik byte kód összehasonlítása.*

4.5.3. Viselkedés meghatározása

Az előző lépés végeredményeként rendelkezésünkre áll a memóriakép egy szelete, amely a Dalvik byte kódot tartalmazza. Ebből a fájlból az összes string-et kigyűjtve - sok más mellett - megkaphatjuk a felhasznált osztályok és kódban előforduló függvényhívások listáját. Amennyiben pedig ismerjük a kódban előforduló rendszerhívásokat, akkor már pontos képet alkothatunk a vizsgált szoftver működéséről.

4.6. Megoldás erősségei és gyengeségei

Az elkészült szolgáltatást több szempontból is lehet vizsgálni. A megoldás előnye, hogy egy humán szakértő számára nagyban megkönnyíti és felgyorsítja a vizsgálatot. A lefuttatott eszközök listája könnyen változtatható konfigurációs fájlok segítségével, így a szoftver könnyen testre szabható. A vizsgálataim során az emulátor alapú megoldást választottam, annak olcsóbb, biztonságosabb és könnyebb skálázhatósága miatt, azonban a tesztek és mérések fizikai eszközön is elvégezhetőek (mint ahogy azt be is mutattam), így a vizsgált kódok kis valószínűséggel detektálják az analízis tényét.

A memóriakép alapú viselkedés vizsgálat több előnnyel is rendelkezik a korábbi megoldásokhoz képest. A rendszerhívásokon alapuló viselkedés analízis egy sokkal részletesebb képet adhat egy szakértőnek, mint például egy jogosultság alapú viselkedés analízis. Nem csak arra derül így fény, hogy milyen szolgáltatásokat használ egy app, de arra is, hogy pontosan hogyan.

Ennek a megoldásnak a segítségével lehetőségünk van olyan kártékony alkalmazásokat is detektálni, amit egy permission alapú rendszer sosem lenne képes. Több helyen is olvasható példa[9] arra, hogyan lehetséges a Java reflection[13] technikát kihasználva olyan szolgáltatásokhoz hozzáférni, amelyekre nem kért az alkalmazás engedélyt. Ezeket a viselkedéseket egy permission alapú megközelítéssel nyilván nem lehet detektálni, az alkalmazásban viszont a viselkedéshez köthető rendszerhívások megtalálhatók.

Egy másik példa a jelen megoldás erősségének a bemutatására a Drive-By-Download[12] technika. Ezen technika lényege, hogy egy alkalmazás futásidőben tölt le kódot az internetről, amelyet aztán futtat is. Ilyenkor nem csak a permission alapú megoldások, de a különböző statikus kód elemzések is haszontalanok, mivel a kártékony kód nem található meg (vagy csak nagyon kis részben) az alkalmazás forrásfájljaiban. Ezzel szemben viszont, miután megtörténik a letöltés és az újonnan érkezett kód is futtatásra kerül, az bekerül a memóriába így az APKAnalyser képes lesz azt is detektálni. Ez a technika ez utóbbi időben egyre nagyobb számban jelenik meg[4], így ennek detekciójára mindenképp érdemes hangsúlyt fektetni a jövőben.

Gyengeségként mindenképp meg kell említeni, hogy egy memóriakép készítésével olyan mintha egy fényképet készítenénk a rendszerről. Azt nem tudjuk, hogy pontosan hol tart a futás, csak a betöltött kódot tudjuk elemezni. Ebből kifolyólag abban sem lehetünk teljesen biztosak, hogy a megtalált kód valaha is le fog futni. Ennek ellenére, ha egy alkalmazás kártékony viselkedésre utaló kódot tartalmaz, még akkor is ha azt nem futtatja le, óvatosan kezelendő.

5. fejezet

Eredmények és Továbbfejlesztés

5.1. Vizsgálatok eredménye

A rendszert a fejlesztés során és végén is több lépésben teszteltem. Az UkatemiSHIELD alkalmazás tesztelését több szempontból is elvégeztük: mint mobil alkalmazást megvizsgáltuk, hogy az adott környezetnek megfelelően működik-e, illetve mint malware detekciós eszközt vizsgáltuk a hatékonyságát. Ezeket a teszteket a 3.7 részben mutattam be részletesebben.

A kutatás során új detekciós eljárásnak számító APKAnalyser tesztelésére több figyelmet fordítottam. A rendszert két szempontból vizsgáltam meg. Egyszer ellenőriztem, hogy a módszer segítségével mennyire hatékonyan lehet a legitim alkalmazásokat a kártékonytól megkülönböztetni. A második tesztelés során pedig azt vizsgáltam, hogy mennyire lehetséges egy adott malware családhoz tartozó alkalmazást felismerni vele.

Az APKAnalyser működése befolyásolható a konfigurációs fájlokon keresztül. Lehetőségünk van a detekciós módszeren is változtatni: megadhatjuk, hogy melyik rendszerhívások jelenlétét keresse a rendszer, illetve, hogy azok megléte mekkora súllyal szerepeljen. A hívások listája mellett egy másik állítható paraméter az a kártékonyság meghatározásának határa, vagyis, hogy mekkora érték felett döntsön a rendszer afelé, hogy egy kód már kártékony. A két paraméter természetesen összefügg. Ha több általánosabb függvényhívást adunk meg detekciós feltételnek, akkor a minden alkalmazáshoz rendelt detekciós érték is magasabb lesz általánosan, így a malware küszöböt is feljebb kell helyeznünk. A megfelelő lista és érték megadása a vizsgálat egy kulcs lépése, így ennek meghatározására különös figyelmet kell szánni.

Egy minta megvizsgálásának az átlagos ideje 10-12 percnyi időbe telt, ezért összesen csak néhány száz mérést tudtam végezni.

5.1.1. Malware detekció tesztelése

A tesztelés során több száz kártékony és tiszta alkalmazás közül választottam véletlenszerűen mintákat. A tiszta alkalmazásoknál arra figyeltem, hogy a VirusTotal eredménye mindegyik mintának 0 találatot adjon vissza. Ugyanezen a módszerrel választottam mintákat a kártékony halmazból is, itt természetesen a bekerülés feltétel a VirusTotal-on mért nem 0 detekciós arány volt.

A tiszta mintáknak a pontos eredményei a Függelék F.3 részében találhatóak. Ugyanígy a kártékony kódok detekciós eredményei pedig a Függelék F.4 részében találhatóak.

A mérések során előfordult, hogy egy alkalmazás vizsgálata nem sikerült. Ilyen esetekben általában az APKTool nem volt képes a telepítendő mintának a csomag nevét meghatározni. Ezeket a vizsgálatokat kivettem az eredményekből, mivel ilyen esetben nem a detekciós módszer hatékonysága jelenne meg az eredményben. A mérésemben így 56 db tiszta és 55 db fertőzött minta szerepelt.

A mérés során kiválasztottam néhány gyanús rendszerhívást, amelyek jelenlétét vizsgáltam. A rendszerhívások listája a következő volt:

```
"dangerous_system_calls": [
    "Landroid/telephony/SmsMessage",
    "Landroid/telephony/SmsManager",
    "getServiceCenterAddress",
    "sendMessage",
    "android.intent.action.SMS_SENT",
    "ACTION_SENT_SMS",
    "Landroid/provider/Telephony$Sms$Intents",
    "Landroid/provider/Telephony",
    "getDisplayOriginatingAddress",
    "getSMSTempBlockNumAndTimes",
    "Landroid/os/Message",
    "Landroid/location/LocationListener",
    "Landroid/location/Criteria",
    "Landroid/location/Location",
    "android.intent.action.CALL",
    "Lcom/android/internal/telephony/PhoneConstants",
    "Landroid/telephony/CellInfoWcdma",
    "Landroid/telephony/CellSignalStrength",
    "Landroid/telephony/PhoneStateListener",
    "Landroid/telephony/TelephonyManager",
    "Landroid/telephony/CellLocation",
    "Landroid/telephony/gsm/GsmCellLocation",
    "Lcom/android/internal/telephony/",
    "Landroid/telephony/",
    "com.android.internal.telephony.",
    "telephony.sms.receive",
    "telephony.sms.send",
    "getPhonenumber"
]
```

A listában is látszik, hogy a rendszerhívásoknak a Dalvik byte kód szintű megfelelőjét kell megkeresni, mivel ez szerepel a memóriában. Szerencsére a Dalvik byte

kód ember számára is viszonylag könnyen olvasható, így ezeknek a hívásoknak a megértése nem jelent komolyabb problémát.

Mérésem során minden találatot egyforma értékkel értékeltem, minden találat 1-el növelte a mintához rendelt értéket. A határt a 12-es értéknél húztam meg, így ami az alá esett, azt a rendszer tiszta mintának tekintette, ami a fölé, azt pedig malware-nek.

Ezekkel a beállításokkal a teszt mérésem során 71,4%-os pontosságot sikerült elérném a fertőzött minták vizsgálata során, valamint 73,3%-os pontosságot a tiszta minták során. A mérés teljes pontossága 72,3%, vagyis a rendszer a jelen beállítások mellett 0,723 valószínűséggel helyesen állapítja meg egy mintáról, hogy kártékony-e vagy sem.

5.1.2. Malware család felismerés tesztelése

A keresett hívások listáját megváltoztatva lehetséges egy vizsgálatot kifejezetten egy malware családra szabni. Ebben az esetben a cél a család tagjaihoz tartozó minták azonosítása, vagy például két család megkülönböztetése. Ez utóbbira végeztem egy mérést: a széles körben elterjedt DroidKungFu illetve Lotoor családok mintáit próbáltam megkülönböztetni. A mért pontos értékek a F.6 részben találhatók a DroidKungFu mintákra, míg a F.5 részben találhatóak a Lotoor mintákra.

Az ellenőrzés során a Lotoor mintákat 0,933 valószínűséggel sikerült eltalálni, míg a DroidKungFu mintáknál ez az érték 0,8 volt. Összességében a rendszer 0,88 valószínűséggel helyesen döntött egy minta besorolásáról.

5.2. Továbbfejlesztési lehetőségek

A jelenleg fejlesztői béta állapotba kiadott Android 5.0 (Lollipop) verzió már alapértelmezetten az új futtatókörnyezetet fogja használni. Az ART (Android Runtime)[7] már az Android eggyel korábbi verziójában is elérhető volt tesztelésre, azonban a felhasználók csak egy kis százaléka próbálta ki.

Az új futtatókörnyezet előnye, hogy az alkalmazások immáron nem egy virtuális gépben futnak, hanem egyszeri lefordítás után az eszközön már a gépi kód tárolódik, így az alkalmazások már natívan képesek végrehajtódni. Az új környezet előnye egyértelműen a gyorsulás, azonban így új biztonsági problémák is felmerülnek.

Az új runtime következtében az alkalmazások rendszerhívásai nem lesznek ilyen könnyen kiolvashatóak a memóriából. Ezután már csak lefordított gépi kód lesz a memóriában, így annak elemzése újfajta megközelítést fog igényelni. A rendszer egy következő továbbfejlesztése lehet, hogy az új futtatókörnyezetet is támogassa.

Az új Android verziók a múltban sem terjedtek komolyabb sebességgel, mivel a

készülék gyártók csak nagyon lassan vagy egyáltalán nem adják ki a frissítéseket. Ebből kifolyólag bár az új ART lesz az alapértelmezett az Android 5.0-ban a "régi" Dalvik környezeten alapuló megoldásom is valószínűleg még évekig hatékony tud maradni, mielőtt megjelennek a kifejezetten az új rendszert támadó malwarek.

Irodalomjegyzék

[1] *Android Mind Reading: Memory Acquisition and Analysis with DMD and Volatility*, 2012.

[2] Balogh György Balázs László. Static malware analysis with statistical methods, 2014.

[3] Robert C. Brodbeck. Covert android rootkit detection: Evaluating linux kernel level rootkits on the android operating system, Junius 2012.

[4] Fortinet. New drive-by download android malware. `http://blog.fortinet.com/post/new-drive-by-download-android-malware`.

[5] Volatility Foundation. Volatility framework. `http://www.volatilityfoundation.org`.

[6] Google. Android emulator goldfish kernel. `https://source.android.com/source/building-kernels.html`.

[7] Google. Android runtime. `https://source.android.com/devices/tech/dalvik/art.html`.

[8] Google. Ui/application exerciser monkey. `http://developer.android.com/tools/help/monkey.html`.

[9] Intrepidious Group. Java reflection in android... ftw. `https://intrepidusgroup.com/insight/2012/04/java-reflection-in-android-ftw/`.

[10] International Data Corporation (IDC). Apple cedes market share in smartphone operating system market as android surges and windows phone gains. `http://www.idc.com/getdoc.jsp?containerId=prUS24257413`.

[11] Google Inc. System permissions. `http://developer.android.com/guide/topics/security/permissions.html`.

[12] Microsoft. What you should know about drive-by download attacks.
`http://blogs.microsoft.com/cybertrust/2011/12/08/what-you-should-know-about-drive-by-download-attacks-part-1/`.

[13] Oracle. Java reflection.
`http://docs.oracle.com/javase/tutorial/reflect/`.

[14] Kaspersky Lab Global Research and Analysis Tema (GREAT). Kaspersky security bulletin. 2013.

[15] Borja Sanz, Igor Santo, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. Puma: Permission usage to detect malware in android, 2012.

[16] Bhaskar Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, and Cristina Nita-Rotaru. Android permissions: A perspective combining risks and benefits. 2012.

[17] Sony. Sony Xperia Arc S.
`http://www.sonymobile.com/global-en/products/phones/xperia-arc-s/`.

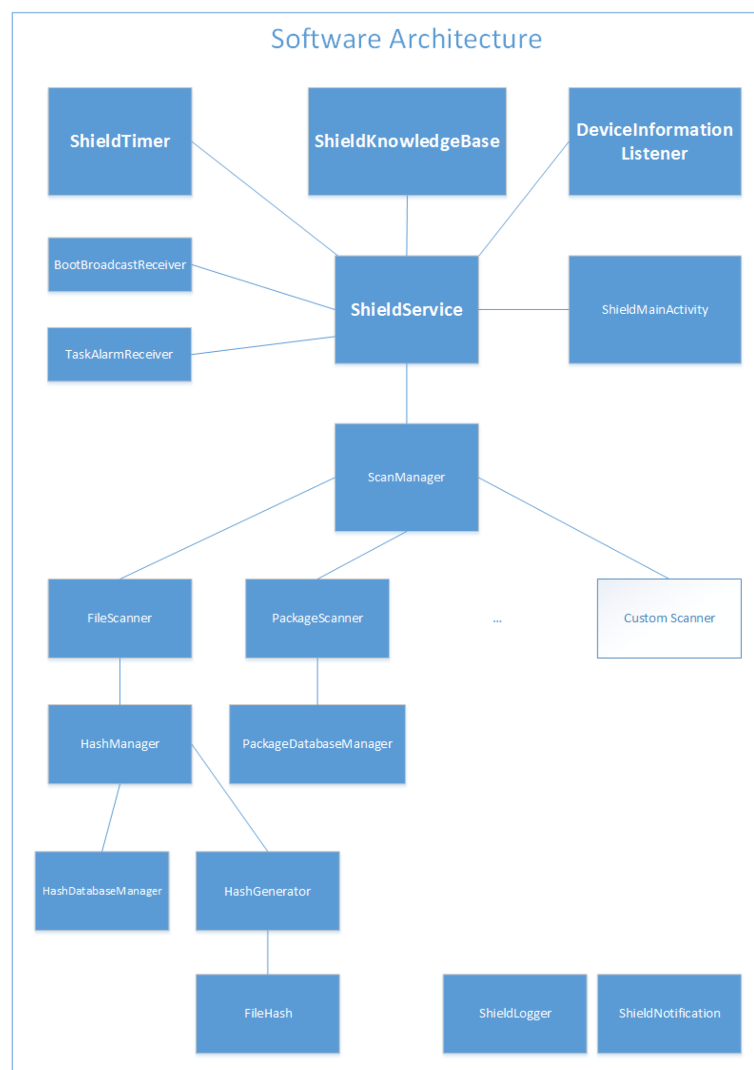
[18] Váradi Szabolcs. Kártékony kódok detektálása android platformon, 2014.

[19] Connor Tumbleson. Android-apktool.
`https://code.google.com/p/android-apktool/`.

[20] AAron Walters and Jr. Nick L. Petroni. Volatools: Integrating volatile memory forensics into the digital investigation process. In *BlackHat DC*, 2007.

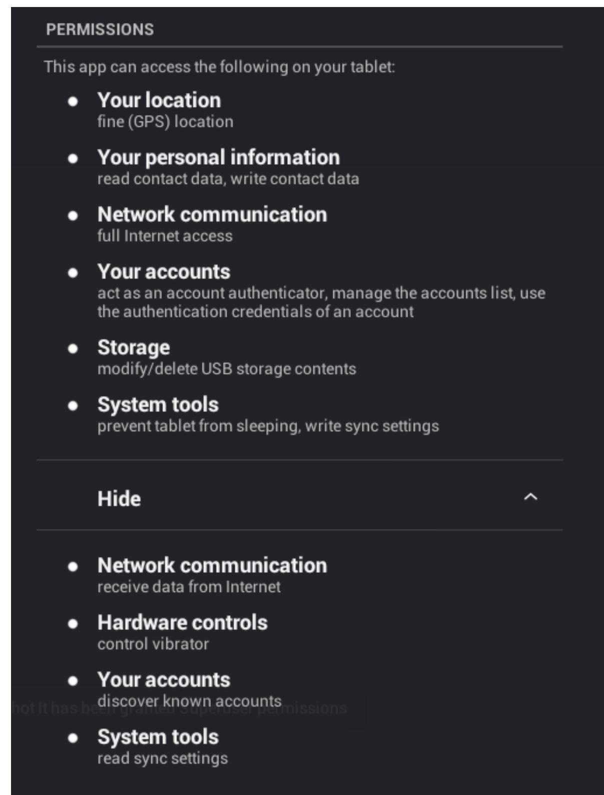
Függelék

F.1. Az UkatemiSHIELD alkalmazás szoftver architektúrája



F.1.1. ábra. *Szoftver architektúra áttekintése*

F.2. Leggyakoribb gyanús Android jogosultságok listája



F.2.1. ábra. *Jogosultságok[18]*

Az Androidon az alkalmazások a jogosultsági rendszer segítségével érik el az operációs rendszer nyújtotta adatokat, felhasználói adatokat, illetve a telefonba épített hardverek szolgáltatásait, például a GPS, gyorsulásmérő, Wi-Fi. Egy alkalmazás telepítésekor a rendszer arra kéri a felhasználót, hogy adjon engedélyt neki a felsorolt különböző tevékenységekhez szükséges jogokhoz. Erre csak telepítéskor van lehetőségünk, később már nem lehet megtiltani egy alkalmazásnak, hogy ne férjen hozzá a kért szolgáltatásokhoz. Sok fejlesztő nem törődik ezzel és egy megszokott jogosultsági listával dolgozik minden egyes alkalmazásánál, annak ellenére, hogy lehet, a jogosultságok több mint felére nincs is szüksége az adott szoftvernek. Nagyon sok jogosultság van az Android rendszerben, és ezek tartalmaznak olyat is, amelyek a felhasználó tudta nélkül hajthatnak végre SMS küldést, hívásindítást. A következőekben a fontosabb jogosultságokat gyűjtöttem össze, amelyeket a támadók előszeretettel használnak a programjukban. [11][16]

ACCESS_FINE_LOCATION

Engedély arra, hogy egy alkalmazás hozzáférjen a precíz helyzetéhez. Ez azt jelenti, hogy az eszköz által szolgáltatott összes forrást igénybe lehet venni a helyzet meghatározásához. Ezek a GPS, Wi-Fi és a cellainformációk lehetnek.

ACCESS_NETWORK_STATE

Engedélyt ad egy alkalmazásnak arra, hogy hálózati információkhoz hozzáférjen.

CALL_PHONE

Engedély arra, hogy hívást kezdeményezzünk úgy, hogy az nem megy keresztül a tárcsázó felhasználói felületén, és a felhasználónak nem kell jóváhagyni a hívást. Ezzel a jogosultsággal automatizáltan tudunk hívást indítani.

CALL_PRIVILEGED

Engedély az alkalmazásnak arra, hogy bármilyen telefonszámot tudjunk tárcsázni. Ebbe beletartoznak a vészhívó számok is. Az előző jogosultsághoz hasonlóan ez sem megy keresztül a felhasználói felületén a tárcsázónak és nem kell jóváhagyást adnia a felhasználónak a hívásra.

CHANGE_WIFI_STATE

Engedély egy alkalmazásnak, hogy megváltoztassa a hálózati kapcsolat állapotát.

INSTALL_PACKAGES

Engedély egy alkalmazásnak, hogy más alkalmazás csomagokat telepítsen.

INTERNET

Hálózati socketek nyitását hagyja jóvá az alkalmazásnak.

MASTER_CLEAR

Gyári beállítások visszaállítása. A telefonról törölhet minden információt.

READ_PROFILE

A felhasználó személyes információit tudjuk olvasni ennek a jogosultságnak a segítségével.

READ_PHONE_STATE

Csak olvasásra hozzáférést biztosít a telefon állapotához. Arra használják, hogy detektáljanak egy bejövő hívást, illetve azt, hogy a telefon egyedi azonosítóit kiolvassák, ami lehet az IMEI szám.

READ_CONTACTS

A telefonkönyv adatait tudjuk olvasni a jogosultság segítségével

READ_CALL_LOG

A hívási előzményekhez kapunk hozzáférést ezzel a jogosultsággal.

PROCESS_OUTGOING_CALLS
: Engedély arra, hogy megfigyeljünk, módosítsunk illetve megszakítsunk kimenő hívásokat.

READ_CALENDAR
: Engedély, hogy olvashassuk a felhasználó naptárát.

READ_SMS
: A felhasználó szöveges üzeneteit olvashatjuk ennek a jogosultságnak a segítségével.

RECEIVE_BOOT_COMPLETED
: Engedély, hogy az alkalmazásunk elkaphassa az ACTION_BOOT_COMPLETED broadcast üzenetet, miután a rendszer befejezte a bootolást. Így nincs szükség arra, hogy a felhasználó elindítsa az alkalmazást, hanem a telefon bootolása után automatikusan indíthatjuk a szolgáltatásainkat.

RECEIVE_SMS
: Ennek a jogosultságnak a segítségével az alkalmazás fogadni tudja a bejövő SMS-ket és bármilyen műveletet tud végezni vele. Beállítható egy prioritási sorrend, hogy melyik legyen az első alkalmazás a sorban, aki megkapja az SMS-ket. A malwarek próbálják ezt kihasználva az első helyre beregisztrálni magukat. A szakdolgozat írása közben jelent meg az Android legújabb verziója a 4.4 KitKat, amiben már nem lehet megkerülni a beépített SMS alkalmazást és a sorban csak mögé lehet feliratkozni.

SEND_SMS
: Megengedi egy alkalmazásnak, hogy szöveges üzenetet küldjön. Az új Android verzióban ez az üzenet szintén meg fog jelenni a gyári SMS alkalmazásban.

WRITE_CONTACTS
: Írási engedély a felhasználó telefonkönyvéhez (olvasni nem tudjuk vele).

WRITE_CALENDAR
: Írási engedély a felhasználó naptárához (olvasni szintén nem tudjuk vele).

WRITE_SMS
: Engedély arra, hogy szöveges üzenetet tudjon írni egy alkalmazás.

WRITE_SETTINGS
: Engedély arra, hogy írni és olvasni tudjuk a rendszerbeállításokat.

F.3. Tiszta Android alkalmazások detekciós értékei

```
0483c547403d3f815b4629c20489b1c2d608c60: 0,
06261ab3c6e6894cda487ab224418950ff60fd07: 12,
06b87099e303d9bbcb561b34e6ee4c26ce69b817: 1,
0857eb82bb7b65b4c13a8b0d1b5d2aaea7a81d24: 3,
097893e23444b9d9dfb3c68ac65ad9299aea50ac: 13,
142fb7e9eeef2760570c329798ebb51b9735b2ff: 4,
178c386bda81d7f20c3d493f4fed2424af3030d4: 13,
18de0405e906f6b7d6ba8f7ce166e93f1474dc99: 6,
1b6625e4163bb1bb347a0f4aee552981c7d2ac42: 4,
21a5891ea0bdeef47ba215bf9b3550fce2919a51: 12,
23fb6178e956996a709264b093a59a5d64c5cf1e: 10,
23fdbb79aa9d98c133cf1b0628d32a90b5799edd: 4,
243a55fdb0ff2352f33742c81be8d6b7ed743281: 16,
24993565820d7f73ef0142a1fef205fc567b8a57: 10,
267cfbdc2a79b35b9033e215e984c06e313124b9: 0,
341431444e6c439f58bcec5a61e83bc32738969f: 13,
3624c36846581cded21c82133aa5f2384c6af301: 4,
37e81405853f6299b97dcc2c7bd186c9bbf95e13: 2,
3a73fade4d2519b6c39971aa39323ca3345993bd: 2,
3db9a4edd3f6ead8128fb74a0eec7c27a77db58a: 12,
4dbb128f281578dc6b960fbadd2917133a501616: 1,
542de9f9916e064fcca51b359b64a0b4edae920d: 15,
5bcf8050f2a521f910e6204af961c2c434d4d9e3: 10,
6dcff5fff3476185e7273600eb752f2074cdcd8d: 13,
75af8d9cc0e88b9a178e29a473ab3db7dac11275: 13,
7ee1e65fe01a61b30b8f39642030b2250cfe3d43: 4,
7fdbd49918ba8e4b4db1a8fc57f49ce762902d7c: 4,
859968c3fcd2327b6456762c894caf053fc7c46a: 2,
85aeca32e2db278314b34ae624af63ca8c3e6bb9: 15,
8b53d369cf22ecb499b24480d92818cd11ff8bba: 14,
8c670548da5376a5fbff9f8c417d2fc52a1718cd: 12,
8d1918c502839f0720e15d5b4b67c4424e0d073e: 4,
9038bc53ff201380a18cc7f994bb6c933c0007e6: 0,
9169567e6c8aabaa913285c825f79dc7cbfcdecd: 5,
9e50775ecd26b03b71ee8651319daeae193e31c4: 4,
9f27f76e828f341341d368df21c1a5c9ea3dcc14: 18,
a01ea750a1bff4eb61d64101568e58a228542493: 2,
a7d2370c49219ed9208d780dc529ce6e84b4ec74: 13,
a9ff9a94a3d01e48ccf4658e1d725909aa51c6cd: 1,
ae7c77defdb1735ac2917b947a551f96787f8e69: 0,
b0886f3f2fcb3ec78d874ad5135e0682d3a7490e: 14,
b4747f48d60403c52dedaf360bf6aeccaff173e9: 4,
b768a2de8b2465552afff5fd84c001d906202c5b: 16,
b8a49e29580735bbec0c1eb09dd15e8c14530219: 4,
bbfdc111fa6baf4156cdadb4a9905e7d64205367: 13,
bca68232ce7a2e2255ac756343922a0c8f5318d1: 0,
ca4ed6588c8355eb6aa5bddd7ecbf0f53651a480: 0,
d15302131abad7a0c222edb15bb56d47f3a25ce9: 10,
df19f44de7aaeb944c2cf1a01c13bacd861ae918: 13,
e092fa8718a02a3326d33ba5107630e510ad9b42: 9,
eb401ce6984dca4fc07f1ccd52d4744c1e484cf0: 11,
ed8065c85e0e46b8896b7107d98681ef5fb68190: 4,
f05f7a51b26952821bac68e9c565b7cab5be8702: 9,
f2499b2248ac6b14132b7813b64ce0dcb28d1cb8: 9,
f33ba530ff318a41d79a6740906cc8da66c38d14: 7,
f5a137cbd25929abcb9b21363967841bb378d44a: 4,
```

```
fd4af97787677a373f8dc6f977f434b3c0f10437: 2
```

F.4. Kártékony Android alkalmazások detekciós értékei

```
005165973a9b1859c39ca753f8df8d436f73de66: 5,
025a0a93f4796e05c454bc181b8fd059245de539: 22,
0274a66cd43a39151c39b6c940cf99b459344e3a: 11,
0874540015f36d46973b684fcce14ec705b1b9e4: 18,
08a21de6b70f584ceddbe803ae12d79a33d33b50: 14,
0936b366cbc39a9a60e254a05671088c84bd847e: 5,
0b6cac1d3b2408748afaf5dbe807f29ef4e0375b: 20,
16706fe77bf2fa73ef2d85543ae1839772d429f9: 14,
17144b0e95a07ffd5bd7c8e3bf95004fe5fe2305: 32,
18ea5584ffb185baf2bb5a87324ae46f7e40ac33: 12,
1c0a6b1c5d24cbba9b11020231fffc0840dd7e10: 34,
24db6f496e87f038b48867808c51c830a6264517: 24,
25b55588a296a58191fd2daa6de2aab3951eb99d: 31,
2c8d75b4de0a7415eeb24ae0b6cbe04392d60e3b: 16,
2cfa26bb22bbdc4e310728736328bde16a69d6b4: 17,
2e058566681bd767edd930fce75114d4612db994: 8,
35338a58e68de0d835d7cb49db82215ca5ebbbda: 11,
35969ef00dc29bf7d8d3a5b9c436c087985d36b9: 33,
35b223e521abc1cb6b8043f95c2a133c11ed8be4: 22,
3af9b2d73ec08af9226d0f24ee6a2d41d9023bb1: 18,
3d451f7093cd47ecf9e41317221569322b8acd20: 5,
4147f7d801c4bc5241536886309d507c5124fe3b: 17,
46454396937f389cf2e438b7047a27350b9f019e: 13,
4de1730332ac35e99337c78ab9aee4fc93f71fc0: 14,
4e37279fb5c494c6c0e26e54046d43b7d57d2c29: 34,
5b7146ef3177fa491d1720a7df46f3f6d40a2799: 27,
5ba9011c8945d29fde942e5f914c912a04e00bca: 15,
5c70988fc9751f283b2f2b5733f9e7f2a54afa69: 16,
5cea3de233f4527b5eb582e17c6f2eb396f9bcf6: 24,
5d6e726eae1ec333363a0308f4fee075245690c9: 22,
5d7a680c041171df0a4a227551e1b2b7c5cc69af: 19,
5d9721692031af2a6a05cb55e54d6687eb408367: 11,
5e2fb0bef9048f56e461c746b6a644762f0b0b54: 16,
619389e71656f3198ede0b249c45455898d60483: 19,
63e642f0d859e096342321c9e03baca7cd1210fa: 13,
657d6567d102d4d2185a4d9a9874d460bfb052c6: 33,
6a5f31a1da2c0956fd49ea27ab45f777c75ad90c: 26,
89225d8c8dcc4c81789c3958e07a51e22fc94af5: 14,
8f85dbb8f3b58c40d1c5cabe2f72f7a9480a460f: 12,
8fc445ba6e8ef561607a41fc83008f92890a026f: 15,
9440bb3da5e1ad862f357248b5da0c59dc7fc96b: 8,
94b56252ff610126135c568b1cc7b92405b9e608: 5,
98057097b1810c2a08a4662d11da3f8fe1bc6ef0: 17,
988769f06f927f554d76a52767c80e9f7c9b2158: 9,
a0f141c4464dbe0c949945b0d4bca7117f032b9f: 12,
a1fa9de17c36f00fbbdfffc9bfc3c858b9202f73: 14,
a7f94d45c7e1de8033db7f064189f89e82ac12c1: 19,
b349851d2a8ba476a0099c17714559f713aa2fdc: 13,
b6d0cf473002347796a460b80209aacb0ce4f5a3: 37,
b78209c5da4eb84b6afccdb4a6690dd684812bbd: 9,
bc2dedad0507a916604f86167a9fa306939e2080: 21,
bd7e85f5a0c39a9aeecc05dbc99a9e5c52150ba6: 17,
c6b7ec91f6e237978552a478306fb6e01c9f15e9: 12,
c9368c3edbcfa0bf443e060f093c300796b14673: 24,
```

```
e0d68c0e21eeeca1f8718e36749506b5ad9d96e4: 14,
f1d8b11012df9b898ca2f9b0a5a97ef79b8a5e1d: 9,
```

F.5. A Lotoor malware család detekciós értékei

```
145ccd42d0944730579a77ddd0a6013bfb3feaf5: 3,
152c2790f469b6fad7ee6e785f15666cf951ad06: 10,
1bde84f283addbe925608090bbf1d0977f632f35: 3,
2c4fc7bf5db55f41c5dab938a780edcc26a72496: 1,
50e868897b716ad8aeb201fd9bcfa62eb035aeeb: 8,
5513275554d41a307d0dad8037f693d7d50cdd28: 5,
6311ec7b8f44806f389674ec88d9f668616b83e8: 2,
75d28d2ea7526d2685c4eac187aaa675bcde934c: 3,
89d5cbab8a96ea68d35bed85c90021be3b257adb: 1,
8fdbb9790359e81fa9a9ec3ba04da8785775e457: 5,
92e2d849166ca23c8e77682f6e62ce82b7001283: 13,
a81622aedae6a8db930a9a0adae7b0495c8a1ee6: 1,
c1be4543908191ac1d9b6f0a6659ee322e2fae88: 10,
d47740bdf9d8d9e713405e5b33e3085d5bc4de70: 1,
db7576449f7b5f8a1956c4d1f9190dfd4edb4e16: 8
```

F.6. A DroidKungFu malware család detekciós értékei

```
2b889919deec496060a717770c8660803e2f6bd3: 11,
34fcaf82a639f8db510126489f8df1c5be9467f7: 26,
43b049a8e3d0093c15094d842f8d2654bb5e8d1e: 18,
4a5659dfda8ae02fe6328d51c6e5bfc1949db79d: 22,
68392ef467a62903d0191570d714a8bad1682bf9: 16,
7c6265abfce9d634abcf286893f21ff5efa14f97: 20,
a05ee30779430bd171bc0d68137c21a1e03d4af6: 22,
aabc0f19c0b7329d420f4d0c52ea26b777f4e66c: 25,
ac87f85b83cae09dffe221f19c62596726bd3b57: 10,
f95f4cc979d56485a6e58a3d47c23c3c3d96e906: 18
```