Budapesti Műszaki és Gazdaságtudományi Egyetem

Applikációk közötti kommunikáción alapuló sérülékenységek vizsgálata Android operációs rendszeren

TDK dolgozat

Szerző:
Juhász Ferenc

Mérnök informatikus képzés
2. évfolyam

Konzulens:
Holczer Tamás

Tudományos segédmunkatárs
BME-HIT

Budapest, 2012

Tartalomjegyzék

Köszönetnyilvánítás

Szeretném megragadni az alkalmat, hogy köszönetet mondjak Holczer Tamás konzulensemnek, a BME Híradástechnikai Tanszék munkatársának, aki nélkül nem jöhetett volna létre ez a dolgozat. Köszönöm áldozatos munkáját, nélkülözhetetlen tanácsait, folyamatos támogatását, segítőkészségét, dolgozatom alapos és kritikus szemrevételezését.

1. Bevezetés

1.1. A dolgozat témája, felépítése

Napjainkban az Android operációs rendszer terjedése egyre nagyobb méreteket ölt. Egy ilyen rendszer esetén kiemelt fontosságú a személyes adatok biztonsága, hiszen napról napra nő a potenciális áldozatok száma. Dolgozatomban az Android operációs rendszeren megvalósítható alkalmazások közötti kommunikáció, az Inter-Process Communication (IPC) lehetőségeivel foglalkozom. Az Android biztonsági architektúrájának alapvető része az applikációk szeparációja, ez azt jelenti, hogy alapvetően az alkalmazások működése teljesen elkülönül egymástól, nem férhetnek hozzá egymás fájljaihoz, erőforrásaihoz. Fő célom annak demonstrálása, hogy ez a biztonsági rendszer nem eléggé átgondolt, kiforrott, így a felhasználók személyes adatinak biztonságba veszélybe kerülhet. Mivel egy alkalmazásnak a telepítéskor kell jogosultságot adni különféle rendszererőforrások használatához, és telepítés után nincs ellenőrizve az alkalmazások közötti kommunikáció, olyan szenzitív felhasználói adatok juthatnak el egy applikációhoz, amely adatokhoz jogosultságai alapján nincs hozzáférése.

Dolgozatom elején alapvető áttekintést adok az operációs rendszerről, majd a biztonsági architektúra kerül a fókuszba. Ebben a részben kiemelt fontosságot kap az alkalmazások szétválasztása, mely az IPC lehetőségek egyik fő gátja lehet. Ezt követően a kommunikációs lehetőségek vizsgálata következik. Először az Android fejlesztőkörnyezete által biztosított eszközök működését vizsgáljuk meg. Ide tartozik a gazdag lehetőségekkel bíró Intent, mely rossz kezekbe kerülve felhasználói adatok kiszivárogtatását teszi lehetővé. Ezt követően az úgynevezett sharedUserId rendszerrel is megismerkedünk, amely lehetővé teszi azt, hogy két alkalmazás korlátlanul hozzáférjen például egymás fájljaihoz.

Sor kerül különböző alternatív megoldások vizsgálatára is. Az operációs rendszer alapvetően a Linuxra alapul, így az ezekben a rendszerekben tradicionális eszközök Androidon való viselkedését is meg kell vizsgálnunk. Ide tartozik például a Socket, vagy a Signal, amelyek potenciális veszélyforrásnak számítanak az alkalmazások közötti kommunikáció területén. Ezt követően egy olyan side-channel megoldás is bemutatásra kerül, ahol a Linux eszközeinek felhasználásával, az éppen futó folyamatok monitorozásával valósítható meg kommunikáció két applikáció között. Végül a fájlkezeléssel foglalkozok, ahol először röviden az Android által biztosított fájlelérési megoldásokat vizsgálom meg, majd az applikáció szeparációt megsértő biztonsági hibán alapuló megoldás is bemutatásra kerül: bár a különböző applikációk egymás fájljait nem olvashatják illetve írhatják, ezen fájlok létezésének tényét mégis meg tudják állapítani. Ezt használom ki két alkalmazás közötti kommunikáció megvalósítására.

1.2. Aktualitás

Mivel az eladott Androidot használó készülékek száma napról napra nő, fontos, hogy a biztonsági problémák minél előbb a felszínre kerüljenek, és a megfelelő lépéseket megtegyük a kijavításuk érdekében. Az Android egy nyílt rendszer, bárki fejleszthet rá alkalmazásokat és publikálhatja azokat, így rosszindulatú fejlesztők is jelen vannak a kezdetektől fogva. A dolgozat témája tehát friss, aktuális, talán ennek is köszönhető, hogy jelenleg meglehetősen kevés szakirodalom, publikáció, tudományos cikk áll rendelkezésre az IPC területéről.

1.3. A tesztkörnyezet

Az egyes kommunikációs lehetőségek megvalósítása egy HTC One V típusú telefonon, Android 4.0 operációs rendszeren történt, a fejlesztés során a 15-ös API-t használtam, minden tesztadat erre a készülékre vonatkozik. A telefon a hivatalos gyártói frissítésektől eltekintve teljesen gyári állapotú, alapvető rendszerbeli felépítésében módosítás nem történt, root jogosultság megszerzésére nem került sor. Nyilvánvalóan más készülékeken eltérések lehetnek, elsősorban a sebességadatok esetén tapasztalható más érték a különböző hardveri felépítésből fakadóan, azonban a megoldás megvalósításának tényén mindez nem változtat, márpedig a dolgozat fő célja a lehetőségek felderítése, nem a leggyorsabb, legoptimálisabb kommunikáció megvalósítása.

A dolgozat felépítése tehát a következő: a 2. fejezetben az Androidról, illetve annak biztonsági architektúrájáról adok általános leírást. Ezt követően a 3. fejezetben az Android SDK eszközei segítségével megvalósítható IPC lehetőségeket vizsgálom, ide tartozik az Intent és a sharedUserId. A 4. fejezetben alternatív kommunikációs lehetőségek vizsgálatát, megvalósítását végzem el. Végül az 5. fejezetben a bemutatott hibák, hiányosságok javítására adok javaslatokat, majd a dolgozat összegzése, a következtetések levonása következik.

2. Az Android operációs rendszer

2.1. Az Android általános leírása

2.1.1. Története, jellemzők

Az Android egy Linux alapú operációs rendszer, elsődlegesen érintőképernyős telefonok és táblagépek működtetésére. Eleinte az Android Inc. fejlesztette, ezt 2005-ben felvásárolta a Google, majd 2007-ben az Open Handset Alliance segítségével nyílt rendszerré vált a mobil eszközök számára. Az Android egy nyílt forráskódú projekt, melyet az Apache License alatt hoz nyilvánosságra a Google [2]. A projekt fő célja az operációs rendszer fejlesztése, fokozatos bővítése illetve karbantartása. Mivel nyílt rendszerről van szó, nagy fejlesztőtáborral rendelkezik az Android: az általuk megírt applikációk segítségével új funkciókkal egészíthetik ki a felhasználók a telefonjaikat, a különböző hardverelemeket minél széleskörűbben kihasználhatják, valamint tetszésük szerint testre szabhatják az eszközöket. Ezek az applikációk nagyrészt Java nyelven íródnak, és például a Google Play rendszeren érhetőek el a felhasználók számára.

Az Android rendszer népszerűségét mutatja a tény, hogy 2012 szeptemberében több mint 675000 applikáció volt elérhető a Google Playen, és ebben az időszakban mintegy 25 milliárd applikáció került letöltésre [4]. Az első Androidos telefont 2008 októberében adták el, és mindössze két év elteltével, 2010 végére már piacvezető lett az okostelefonok területén. 2012 második negyedévében 68%-át birtokolta a piacnak, 500 millió aktivált készülékkel, és folyamatos növekedése nem állt meg: napi mintegy 1,3 millió további készülék került eladásra [2]. Bár az Android alapvetően okostelefonok és tabletek számára készült, nyílt mivolta következtében más eszközökön, például laptopokon, ebook olvasókon is használhatóvá vált. Később olyan eszközökön is megjelent, mint a karórák, fejhallgatók, autós CD és DVD olvasók, játékkonzolok, tehát egy nagyon széles körben alkalmazott, népszerű operációs rendszerről van szó. Mára tehát az élet sok területén jelen van, rendkívül széleskörű felhasználóbázissal rendelkezik az operációs rendszer.

2.1.2. Felhasználói interfész

Az Android felhasználói interfésze kevés tényleges nyomógombbal rendelkezik, helyettük főként az érintőképernyőn alapul, amelyen a mindennapokban is használt mozdulatok segítségével lehet kommunikálni a telefonnal: az ujjunk elhúzásával, rövid érintésével, „összecsípéssel” és széthúzással is adhatunk ki különböző parancsokat. Az ezekre a jelzésekre adott válasz összhangban van az érintéssel, tehát nagy sebességű, folyamatos a működés. Speciális hardverek is rendelkezése állnak a mobil készülékekben a felhasználói jelzések feldolgozására, ilyen például a gyorsulásmérő, a giroszkóp, vagy a távolságérzékelő. Az ilyen eszközök biztosítják például azt, hogy a telefon orientációjával legyen összhangban a képernyőn megjelenített tartalom. A rendszer betöltése után a kezdőképernyő fogadja a felhasználót, mely a számítógépek asztalához hasonlít: itt ikonok és widgetek találhatóak meg, előbbiekkel applikációkat indíthatunk el, míg utóbbiak dinamikusan változó tartalommal rendelkeznek, például időjárás-előrejelzésre vagy hírek megjelenítésére lehetnek alkalmasak.

2.1.3. Fejlesztés

Az Android applikációk általában Java nyelven íródnak az Android Software Development Kit (SDK) segítségével. Rendelkezésre áll ezen kívül a Native Development Kit (NDK),

mellyel C illetve C++ nyelven is lehet applikációkat fejleszteni, illetve további rendszerek is elérhetőek alkalmazásfejlesztéshez. Az elkészült applikációk például a Google Playen vagy az Amazon Appstore-on keresztül érhetőek el a felhasználók számára, vagy a megfelelő telepítőfájl letöltésével, melyet egy külső oldalról szerezhetnek be. A Google Play az egyik elsődleges forrás az applikációk megszerzésére. Ez egy online árusítási rendszer, ahol a felhasználók böngészhetnek az alkalmazások között, a fejlesztők vagy a Google ajánlatai közül válogathatnak, és természetesen le is tölthetik az egyes alkalmazásokat. A rendszer automatikusan kiszűri azokat az applikációkat, amelyek a felhasználó készülékével inkompatibilisek, illetve a fejlesztők is korlátozhatják az alkalmazásaik eléréseit üzleti okokból. A Google Playen ingyenes és fizetős alkalmazások egyaránt megtalálhatóak, előbbiekből a Google széles körű applikációbázissal rendelkezik, melyek mind általános (pl. Google Translate), mind speciális (pl. Google Voice) alkalmazásokat tesznek elérhetővé a felhasználók számára, de természetesen a nyílt rendszernek köszönhetően tulajdonképpen bárki elérhetővé teheti alkalmazását.

2.1.4. Az Android és a Linux

Az Android rendszermag a Linux 2.6 (Android 4.0 után 3.x) kernelén alapul, melynek függvényei, programozói eszközei C nyelven íródtak. Az applikációs szoftverek egy keretrendszeren futnak, mely Java-kompatibilis függvénykönyvtárakat tartalmaz. Az Android a Dalvik Virtual Machine-t használja erre, amely a jobb teljesítmény érdekében Dalvik dex-kód (Dalvik Executable) futtatását teszi lehetővé, melyet a Java bytecode-ból állít elő. A platform, amelyen mindez megvalósul, alapvetően az ARM architektúra, azonban támogatott az x86 architektúra is az Android x86 projekt keretében. Az operációs rendszerhez tartozó Linux kernel sok változáson esett keresztül. Alapvetően nem rendelkezik például az összes GNU könyvtárral, melyek segítségével Unix-kompatibilis szoftverek fejlesztésére van lehetőség, így a már létező Linuxos alkalmazások Androidra való áthelyezése komplikálttá válik. A Linux-szal való együttműködés sem volt mindig akadálytalan: bizonyos, Google által implementált eszközök (mint például egy energiagazdálkodási lehetőség, a wakelock) nem kerültek be a Linux kernelekbe, mert a fejlesztők azt kifogásolták, hogy a Google nem tartja karban az általuk megírt kódot. Noha voltak fejlesztői erre a területre specializálva a Google-nek, belső munkatársak szerint az Androidot készítő csapat túl kicsi volt, így az Android fejlesztésével voltak elfoglalva. Később bekerült ez az újítás a Linuxba, azonban kicsit más megvalósításban [2]. A Linuxos alapok miatt az Android rendszeren a flash memória több részre van osztva, például maga az operációs rendszer a "/system" mappában, míg az applikácikók a "/data" mappában találhatóak. Több ilyen könyvtárhoz a felhasználó alapesetben nem rendelkezik hozzáféréssel (nincs root jogosultsága), azonban ezt a jogot meg lehet szerezni biztonsági rések kihasználásával. Különböző Androidos közösségek tagjai előszeretettel teszik ezt telefonjuk még jobb testreszabhatóságának érdekében, azonban ez veszélyekkel is jár, hiszen egyrészt a telefont használhatatlanná lehet tenni, másrészt rosszindulatú alkalmazások fejlesztői is kihasználhatják ezt.

2.1.5. Frissítések

A Google hat-kilenc hónaponként jelentős frissítéseket ad ki. A legutóbbi ilyen frissítés a "Jelly Bean" névvel ellátott Android 4.1. A riválisoperációs rendszerekhez viszonyítva (például iOS) az Androidos készülékekre viszonylag lassan érkeznek meg ezek a frissítések: akár hónapok is eltelhetnek a kiadás és a tényleges telepítés között. Ez részben annak köszönhető, hogy nagyon széles tartományban változik az Androidot használó tele-

fonok hardveres felépítése, és mivel alapvetően e Google készülékére, a Nexusra készülnek a frissítések, így átalakításra szorulnak a különböző készülékek esetén. Nagyon idő- és erőforrásigényes az Androidot a különböző eszközökkel kompatibilissé tenni, így ha egy gyártó úgy dönt, hogy inkább új készülék fejlesztésére fordítja az energiáit, és a régiek frissítésével nem foglalkozik, akkor ezeken a készülékeken általában még akkor sem érhető el egy-egy frissítés, ha azok képesek lennének annak futtatására. Egyesek szerint azért sem támogatják a gyártók a régi készülékek frissítését, mert ez elősegíti az új eszközök vásárlását. 2011-ben a Google számos gyártóval megalkotta a probléma kiküszöbölésére az "Android Update Alliance"-t, amelynek célja a frissítések megfelelő időben való eljuttatása a készülékekre [2].

2.1.6. Licencelés

Az Android egy nyílt forráskódú operációs rendszer. A kód nagy részét az Apache License 2.0 alatt publikálja a Google, míg a Linux kernel változtatásait a GNU General Public License 2 alatt hozza nyilvánosságra. A Linux kernel fejlesztése teljesen publikus módon zajlik, míg az Android többi része zártkörűen, csak az új verziók nyilvánosságra hozatalával. Tipikusan létrehoznak egy új verzióhoz egy új Nexus szériás eszközt, mely az új Androidot támogatja, és a forráskódot ezen eszköz kiadása után teszik elérhetővé. 2011 első részében visszatartotta a Google az egyik frissítését, mely a Honeycomb névre hallgatott és kifejezetten táblagépekre fejlesztették ki. El akarták ugyanis kerülni azt, hogy a gyártók ezt a frissítést mobiltelefonjaik esetén is felhasználják, mivel ebben az esetben csak egy nagyon rossz felhasználói élmény lett volna az eredmény. A forráskód novemberben került nyilvánosságra, a 4.0-ás, telefonokra is ajánlott Android verzió kiadásával [2].

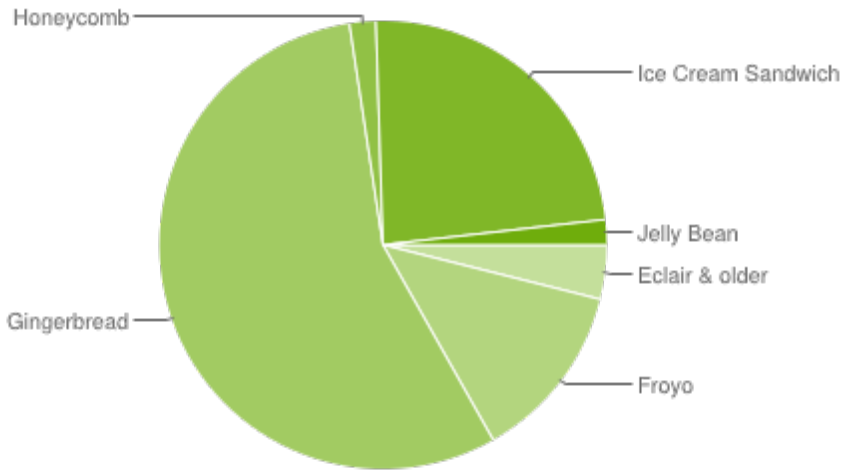
2.1.7. Fogadtatás

Míg 2009-ben az Androidos készülékek mindössze 2,8%-át tették ki a világ okostelefonjainak, 2010-re ez 33% lett, mellyel megszerezték a piacvezető szerepet, 2012-re pedig már 68%-os lett a részesedése az okostelefonok között. 2010 első negyedévében az Android platform az Egyesült Államokban már felülmúlta riválisát, az iPhone-t. Mindez nagyban annak volt köszönhető, hogy az operációs rendszer rendkívül széleskörűen alkalmazható különböző hardvereken. 2011 júliusára, mintegy fél év alatt megkétszereződött az Androidos készülékek eladása, októberre 190 millió eszköz volt a piacon. 2011 decemberében a Pentagon hivatalosan is elfogadta, hogy személyzete Androidos készülékeket is használhasson [2]. Az Androidos eszközök megoszlását mutatja az 1. táblázat az 1. ábrával összhangban, 2012. október 2-ai állás szerint [1].

2.2. Az Android biztonsági architektúrája

2.2.1. A biztonsági rendszer alapjai

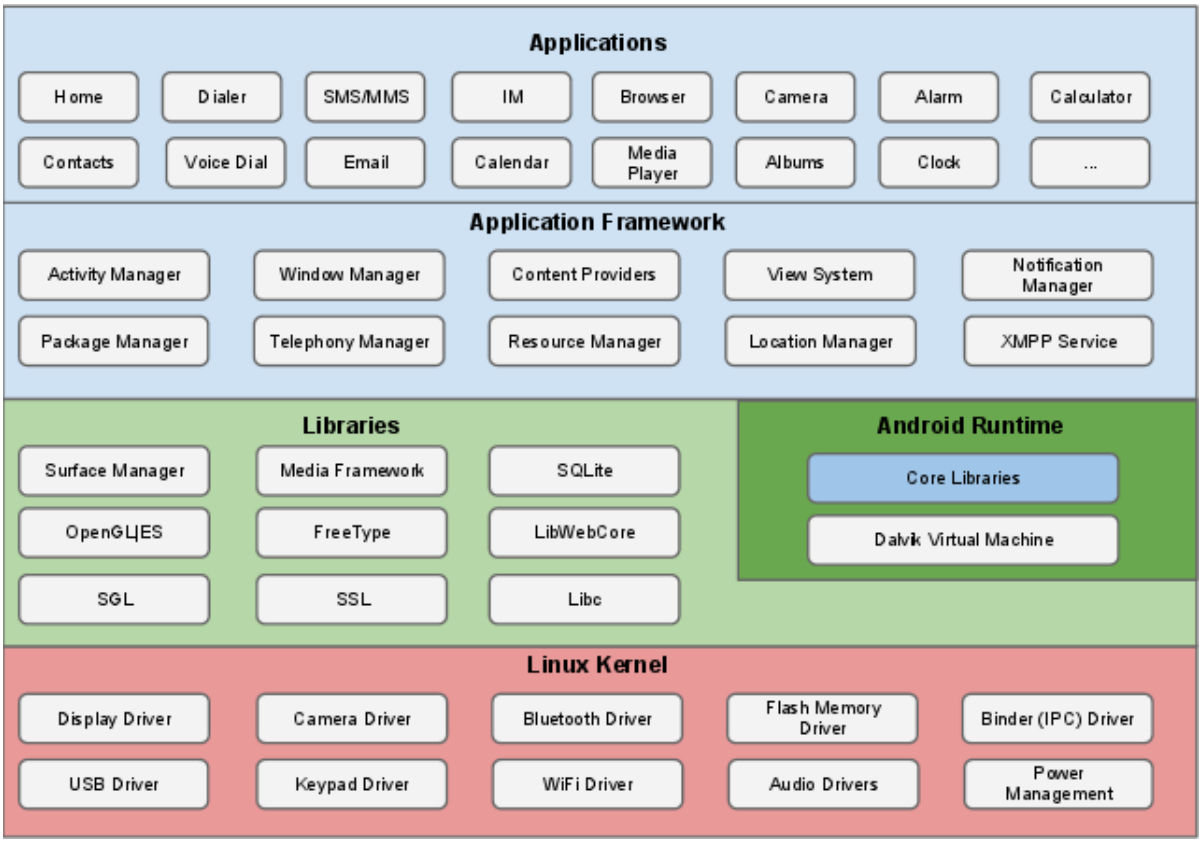
Az Android operációs rendszer alapvető megismerése után lényeges a biztonsági architektúra áttekintése, hiszen a dolgozat célja annak megmutatása, hogy az architektúra bizonyos részei nem eléggé átgondoltak. Az előző részből láthattuk, hogy az operációs rendszer fejlődése meglehetősen nagy mértékű, napról napra több készülék kerül forgalomba, ennek következtében egyre több ember válhat a biztonsági rendszer hibáinak kihasználásából



1. ábra. Az operációs rendszer verzióinak eloszlása

Verzió	**Kiadás dátuma**	**API szint**	**Eloszlás**
4.1.x Jelly Bean	2012. július 9	16	1,8%
4.0.x Ice Cream Sandwich	2011. október 19	14-15	23,7%
3.x.x Honeycomb	2011. február 22	11-13	1,9%
2.3.x Gingerbread	2010. december 6	9-10	55,8%
2.2 Froyo	2010. május 20	8	12,9%
2.0, 2.1 Eclair	2009. október 26	7	3,4%
1.6 Donut	2009. szeptember 15	4	0,4%
1.5 Cupcake	2009. április 30	3	0,1%

1. táblázat. Az Android verziói



2. ábra. Az Android operációs rendszer felépítése

fakadó problémák áldozatául. Gyakran tárolunk a telefonon privát adatokat, nem szeretnénk, hogy bárki belelásson levelezéseinkbe, vagy akár banki tranzakcióinkba. Kiemelten fontos tehát ezen adatok biztonsága, így a biztonsági architektúra megvizsgálása és az esetleges hibák kiszűrése.

Egy ilyen platform biztonságának fenntartása érdekében már a rendszer tervezésénél szigorú szabályok alapján kell létrehozni a biztonsági megoldásokat. Az Android nyílt fejlesztőkörnyezettel rendelkezik, ez azt jelenti, hogy tulajdonképpen bárki fejleszthet alkalmazásokat erre az operációs rendszerre. Nem szabad tehát a fejlesztők kezébe olyan eszközöket adni, amelyek segítségével, felelőtlen használat esetén sérülhet az adatok biztonsága, érdemes ezeket a lehetőségeket már csírájukban elfojtani, hiszen így nem is lesz esély arra, hogy biztonsági rések kihasználását véghezvihessék a fejlesztők. Az Android tervezésénél próbálták mind a fejlesztőket, mind a felhasználókat számításba venni ilyen szempontból: előbbieknek a biztonság megvalósítását alapértelmezett, gyári funkciók, eszközök biztosítják, ezáltal tehermentesítve vannak, míg utóbbiak az applikációk működésének kontrollálásával érhetik el a megfelelő szintű védettséget. Azonban a későbbiekben látni fogjuk, hogy nem mindig sikerült megfelelő megoldás választása a Google részéről.

Az operációs rendszer alapvető felépítése a 2. ábrán látható [3]. Az ábrán látható komponensekre bontás fő célja a megfelelő adatbiztonság megvalósítása, a rendszer különböző erőforrásainak védelme, valamint az applikációk egymástól való elszeparálása. A későbbiekben láthatjuk, miért fontos ez a három tényező, és milyen eszközöket vet be az Android ezek eléréséhez. Az egyes komponensek mindig azt feltételezik, hogy az alattuk lévő szint biztonsága megfelelő. Ahogy az ábrán is láthatjuk, az Android alapja a Linux,

és ez már rögtön magával hordoz egy biztonsági megoldást: az alkalmazások egymástól való elkülönítését. Ez az első, és egyik legfontosabb szintje a biztonságnak, ugyanis ezáltal lehet biztosítani, hogy az alkalmazások egymással ne kommunikáljanak, tehát az IPC (Inter-Process Comunnication) lehetőségek elsődleges gátjává válhat. Ezen rendszer esetén az applikációk egymással való kommunikációs lehetőségeinek a felderítése a személyes adatok biztonságának szempontjából kiemelten fontosak, és mivel a dolgozatom fő célja ezen IPC megoldások felkutatása, ezért kifejezetten erre a területre koncentrál a rendszer bemutatása. A Linux biztosítja tehát a programok szeparációját, ez azt jelenti, hogy egyik alkalmazás nem férhet hozzá sem a fájljaihoz, sem az erőforrásaihoz egy másik applikációnak. Mindezt az úgynevezett felhasználói azonosító (user ID) rendszer segítségével éri el, melynek során minden egyes futó folyamat egy ilyen egyedi azonosítót kap. Így egy "sandboxing"-ot valósít meg a rendszer: minden alkalmazás a saját, és csak a saját erőforrásait, fájljait éri el, csak ezekkel tud gazdálkodni, ezek segítségével tud különböző feladatokat végrehajtani. Nagyon fontos, hogy ez a rendszer tulajdonképpen az Android legalsó szintjén működik, ennek következtében az operációs rendszer teljes egészére kihat: a különböző függvénykönyvtárak, az applikációkat futtató virtuális gép, az összes fejlesztőeszköz efölött a réteg fölött fut, így természetesen ezekre is hatással van ez a megoldás, az applikácók izolációja jelentősen leegyszerűsíti a további biztonsági megoldások kialakítását. A Linux kernel még egy járulékos hatással rendelkezik: mivel natív kód (C, ill. C++ nyelvű) futtatása is lehetséges az operációs rendszeren, ennek ellenőrzését elvégezheti rögtön a legalsó szint, így valóban egy esszenciális részét képezi a Linux az Androidnak.

Érdemes megemlítenünk, hogy nem csak a Linux biztonsági rendszerét, hanem annak tradicionális eljárásait, módszereit is örökölte az Android. Ezek alapján a rendszerben lehetőségünk lehet például a Socketek, vagy a Signalok használatára, amelyekkel applikációk közötti kommunikáció megvalósítása lehetséges, ezekkel az eszközökkel természetesen a továbbiakban foglalkozni fogunk. Láthatjuk tehát, hogy rögtön a rendszer legalsó szintjén egy erős biztonsági megoldás áll, azonban a dolgozat későbbi részében bemutatásra kerül, hogy már ez a rendszer sem tökéletes, és megvalósítható ennek kijátszása.

Most tekintsük a felsőbb szinteket. Az Android fejlesztőkörnyezete több lehetőséget biztosít ilyen típusú kommunikáció létrehozására. Az egyik, és legfontosabb ilyen lehetőség az úgynevezett Intentek használata. Az Intentek, ahogy a nevük is mutatja, egy szándékot fejeznek ki: alapvetően egy másik applikáció, pontosabban annak egy részének meghívását teszik lehetővé. Az Intent hatására a fogadó programrész bizonyos feladatokat hajthat végre (akár a háttérben, a felhasználó tudta nélkül), visszaigazolást küldhet az őt meghívó applikációnak, és adatküldés is megvalósítható a segítségével. Láthatjuk tehát, hogy az applikációk szeparálása sérülhet ennek a fejlesztőeszköznek a használatával, rossz kezekbe kerülve személyes adatok kiszivárgását okozhatják. Az Intentek pontos működéséről, típusairól, korlátozásairól a dolgozat 3. fejezetében részletesen foglalkozunk. Szintén fontos eszköz az ugyancsak Android által biztosított sharedUserID: ennek segítségével a korábban bemutatott Linuxos felhasználói azonosító használata során két különböző applikáció ugyanazzal az azonosítóval rendelkezhet, amely szintén megsérti a sandboxingot. Ahogy az Intenttel, ezzel az eszközzel is a későbbiekben részletesen megismerkedünk.

2.2.2. A Manifest

Minden egyes Androidra fejlesztett alkalmazás tartalmaz egy úgynevezett Manifest fájlt (AndroidManifest.xml). Az operációs rendszer ebből a fájlból határozza meg többek között, hogy az alkalmazás milyen kritikus erőforrásokhoz kér hozzáférési engedélyt. Ez a fájl

tartalmazza továbbá az applikációk egyes komponenseit, egyedi azonosítóját, a futtatáshoz szükséges minimum API szintet, illetve egyéb, az applikációval kapcsolatos járulékos adatokat.

2.2.3. A permission rendszer

Nagyon fontos részét képezik az Android operációs rendszernek a jogosultságok (vagy permission-ök). Amikor egy felhasználó egy applikációt akar telepíteni a készülékére, akkor a operációs rendszer kilistázza, milyen rendszererőforrásokhoz kíván hozzáférni az adott alkalmazás. Amennyiben a felhasználó engedélyt, jogosultságot ad ezek használatára, a telepítés megtörténik, egyéb esetben viszont nem. Ilyen erőforrások például a hálózati szolgáltatás, a szöveges üzenetek küldése, vagy a kamera használata. Láthatjuk tehát, hogy a biztonsági megoldások mind a fejlesztői, mind a felhasználói oldalon jelentkeznek. A jogosultságok azt az érzetet keltik, hogy a felhasználó teljes irányítása alatt tartja a készülékén futó programokat, azok csak az ő által engedélyezett erőforrásokhoz férhetnek hozzá, így kontrollálhatóak az események. A dolgozat fő célja annak megmutatása, hogy ez nincs mindig így. Éppen ez az oka, hogy az IPC megvalósítási lehetőségeit vizsgáljuk: amennyiben az alkalmazások kommunikálhatnak egymással szabadon, tehát adatot tudnak egymásnak átadni, akkor egyik applikációtól a másikig olyan adat is eljuthat, amelyhez a felhasználó nem biztosított engedélyt. Például ha az egyik alkalmazás csak az internet-hozzáféréshez kap engedélyt, míg a másik csak a GPS koordináták lekérdezéséhez, akkor amennyiben kommunikálni tudnának ezek az alkalmazások, a felhasználó földrajzi helyzete egy webszerverre feltölthetővé válna. Ami fontos, hogy a felhasználó ennek nincs tudatában: az alkalmazások telepítésekor az általa biztosított jogosultságok mindezt nem tennék lehetővé, de a biztonsági rendszer kijátszása már igen. Mindez érzékeny személyes adatok nem megfelelő kezekbe való jutását eredményezheti.

2.2.4. Az application signing

Elkészült alkalmazásainkat nem tehetjük elérhetővé rögtön a Google Playen. Minden alkalmazást digitálisan alá kell írni, ezt jelenti a signing. Fontos, hogy ehhez nem szükséges hivatalos hitelesítők használata, minden fejlesztő könnyedén generálhat magának egy kulcsot, és ennek segítségével a fejlesztés befejezésekor aláírhatja, sign-olhatja az alkalmazásait. Ez természetesen egy privát kulcs, csak a fejlesztő ismeri. Mindez azt a célt szolgálja, hogy a fejlesztő egyértelműen azonosítható lesz a publikálás után, valamint azonos aláírással rendelkező alkalmazások egymás erőforrásait felhasználhatják, a megfelelő körülmények között.

3. A fejlesztőkörnyezet által biztosított kommunikációs lehetőségek

3.1. Az Intent

Ebben a részben az Intentekkel foglalkozunk részletesen. Ahogy láthattuk, az Android fejlesztőkörnyezete is biztosít lehetőséget Inter-Process Communication-re, így mindenekelőtt ezek működésének feltérképezésével, viselkedésével, korlátaival ismerkedünk meg, hiszen ezek minden fejlesztő számára hozzáférhető és egyszerűen használható eszközök. A legfontosabb ilyen eszköz pedig az Intent. Segítségükkel képes egy alkalmazás elindítani egy másik applikáció valamely komponensét, ez a komponens így adatokat fogadhat, különböző feladatokat láthat el ezen adatok függvényében, és akár választ is biztosíthat azt őt meghívó alkalmazás számára.

3.1.1. Az applikációk alapvető felépítése

Az Intent küldési formáinak megismerése előtt tekintsük át, mely komponensek indíthatóak el ilyen formán, hiszen ezen komponensek kommunikációját mutatjuk be a későbbiek során. Alapvetően egy alkalmazás négy fő osztályból épülhet fel: ezek az Activity, a Service, a Broadcast Receiver és a Content Provider [1].

Activity. Az Activity nem más, mint egy egyszerű felhasználói interfésszel rendelkező ablak: a felhasználó és az alkalmazás ezen keresztül kommunikál, itt van lehetőség adatok bevitelére, illetve a felhasználó számára információ megjelenítésére. Egy applikáció elindításakor is általában a program alapját képező Main Activity indul el, ezen az interfészen keresztül lehet működtetni a programot.

Service. A Service tulajdonképpen egy grafikus interfész nélküli Activity. Ezek a komponensek a háttérben futva fejtik ki hatásukat, külön szállal rendelkeznek, így egyes időigényes műveletek elvégzésére kiválóan alkalmasak. Ilyen lehet például egy hálózati adatátvitel, amely nagy adatmennyiség esetén sok időbe telhet: ekkor ezt egy Service el tudja végezni, anélkül, hogy a felhasználónak az átvitel végét meg kelljen várnia, így a program működése folyamatos, nincs szükség várakozásra.

Broadcast Receiver. A harmadik nagy a csoport az úgynevezett Broadcast Receiverek csoportja. Ahogy a nevük is mutatja, ők különböző események (broadcast-ok) feldolgozását végzik el. Ilyen esemény lehet egy bejövő hívás, új szöveges üzenet érkezése, az akkumulátor alacsony szintje stb.: ezekről az eseményekről Intent formájában értesülhetnek, és a megfelelő feladatokat láthatják el a különböző történések bekövetkezésekor. Bár felhasználói interfésszel ők sem rendelkeznek, értesítéseket küldhetnek a felhasználónak az értesítősávon keresztül.

Content Provider. A Content Providerek tulajdonképpen speciális adattárolók, melyek a telefon különböző erőforrásainak megosztását teszik lehetővé az applikációkkal. Ilyen erőforrás lehet például a telefonkönyv: megfelelő jogosultsággal bármely applikáció lekérdezheti annak tartalmát. Az adattárolás tulajdonképpen egy adatbázis formában történhet, ebből az adatbázisból lehet lekérdezni adatokat, illetve adatok módosítására is van lehetőség.

A vizsgálatunk szempontjából a leglényegesebb komponens a Service lesz számunkra. A Service segítségével a háttérben dolgozva elrejthetjük a felhasználó elől az adatok átvitelét, valamint azoknak feldolgozását, így észrevétlenül, a felhasználó tudta nélkül leszünk képesek adatok továbbítására applikációk között.

3.1.2. Intentek küldése

Az Intentek segítségével nem csak a komponensek elindítására van lehetőség, hanem adattovábbításra is. Ezt úgy érhetjük el, hogy az általunk egyik komponenstől a másiknak szánt adatokat (melyek tipikusan primitív típusokban tárolt adatok) egy "extra csomagként" csatolhatjuk az Intenthez. Amikor a fogadó fél megkapja az Intentet, akkor indulásakor lekezelheti azt, így megnézheti, kapott-e ilyen csomagot, és ha igen, akkor az elküldött adatokat meg tudja szerezni, azokkal műveleteket tud végezni. Mindez kulcsfontosságú a vizsgálatunk szempontjából: amennyiben két applikáció ilyen formán képes kommunikálni, akkor olyan adatokat tudnak nagyon egyszerű módon egymásnak eljuttatni, amelyek elérésére alapvetően nem lenne jogosultságuk.

Mindezek után tekintsük az Intentek tényleges küldésének folyamatát a különböző komponensek között. A négy alap komponens közül háromnak lehet Intentet küldeni: az Activity-nek, a Service-nek és a Broadcast Receiver-nek. Az Activity-ket, valamint a Service-okat a Manifestben deklarálnunk kell, ahogy ezt korábban is láttuk (a Broadcast Receivereket futásidőben is létrehozhatjuk). A dolgozat szempontjából a gazdag funkcionalitással rendelkező Activity és Service a lényeges, ahhoz, hogy ezek a komponensek Intentet fogadhassanak, mások számára is láthatóvá kell tenni őket az exported flaggel, ezt is a Manifest fájlban tehetjük meg. Mivel számunkra az applikációk közötti kommunikációk a lényegesek, csak az exported tulajdonsággal rendelkező komponenseket vizsgáljuk, hiszen ezek fogadhatnak másoktól kéréseket. Az ilyen komponensek az Intentek mindkét típusával elindíthatóak: ezek az explicit és implicit Intentek.

3.1.3. Explicit Intent

Az explicit Intentek alkalmazása során a célkomponenst egyértelműen megnevezzük. Ez azt jelenti, hogy pontosan megmondjuk, melyik applikáció melyik komponensét szeretnénk elindítani. Természetesen ehhez szükség van az applikáció egyedi azonosítójára, amely egy csomagnév, ez a csomag tartalmazza a komponenseket. Így látható, hogy amennyiben egy alkalmazás felépítését, azonosítóját nem ismerjük, explicit Intenttel nem tudjuk elindítani egy komponensét sem. Azonban ha mi magunk vagyunk a fejlesztői a két alkalmazásnak, ezt minden további nélkül megtehetjük. Rögtön észrevehető, hogy így tulajdonképpen korlátlanul képesek egymással kommunikálni az alkalmazások, valamint folyamatosan tudnak adatot küldeni egymásnak. Így a korábbi példánál maradva, amennyiben a fejlesztő egyik alkalmazása csak az internet-hozzáféréshez kér jogosultságot, míg egy másik alkalmazása a GPS koordinátákhoz (melyek tulajdonképpen két valós számot jelentenek, tehát primitív típusban egyszerűen tárolhatóak és továbbíthatóak), akkor explicit Intenthívások segítségével el tudja ezeket az adatokat juttatni egyik alkalmazástól a másikig, majd feltöltheti mindezt egy webszerverre, mintegy nyomon követve az áldozatot, akinek erről fogalma sincs. Ugyanis ha az Intent egy szolgáltatást indít el, akkor ez a háttérben, észrevétlenül fut, így a felhasználó tudta nélkül végezhet el minden fontos operációt.

3.1.4. Kommunikáció gyakorlati megvalósítása explicit Intentek segítségével

Természetesen mindez a gyakorlatban is kipróbálásra került: két applikációt hoztam létre a fenti specifikáció alapján: az egyik periodikusan lekérdezi a felhasználó koordinátáit, amint változást észlelt, elküldi az új szélességi és hosszúsági koordinátákat explicit Intent segítségével az internet-hozzáféréssel rendelkező alkalmazásnak, ami feltölti az adatokat egy szerverre. Telepítéskor valóban csak a szükséges erőforrásokhoz kértek jogosultságot az applikációk, annak ellenére, hogy egyértelműen látható a kódban: az egyik alkalmazás meghívja a másikat, ráadásul az Intenthez extra adat is járul. Az applikációk futtatása is gond nélkül járt, és hálózati kapcsolat megléte esetén valóban feltöltésre kerülnek az adatok. Utóbbi megoldás egy Service segítségével lett megvalósítva, hiszen - ahogy korábban is írtam - ebben az esetben a felhasználó tulajdonképpen semmit nem vesz észre a folyamatból, az a háttérben futva a maga sebességével feltölti a rendelkezésre álló adatokat, míg a felhasználó bármi mást csinálhat az alkalmazásban.

Felmerült az is lehetőségként, hogy bár telepítéskor nem láthatja a rendszer, hogy a másik applikáció milyen jogosultságokkal rendelkezik (hiszen az első alkalmazás telepítésénél még a második nincs a készüléken), később ez korrigálásra kerül, azonban a telepítés és futtatás után is csak a Manifest fájlban megadott jogosultságokkal rendelkeznek az alkalmazások. Beláthatjuk tehát, hogy egy fejlesztő két applikációjával egyértelműen személyes adatok megszerzésére képes a felhasználó kijátszásán keresztül, mindez az Intentek küldésének és a permission rendszer átgondolatlanságának köszönhető.

3.1.5. Implicit Intent

Ahogy láthattuk, explicit hívásnál pontosan tudnunk kell a hívandó komponens egyedi azonosítóját, mely az alkalmazás csomagnevéből és a komponens nevéből áll. Azonban implicit Intentek esetén egyikre sincsen szükségük, ezek nélkül is elindíthatunk különböző alkalmazásokat, pontosabban azok részeit. Mindezt implicit Intentek segítségével tehetjük meg. Ezek az Intentek azon alapulnak, hogy a Manifest fájlban minden komponens esetén meghatározhatunk úgynevezett Intent-filtereket. Ezen szűrők feladata annak meghatározása, hogy milyen Intentekre kell reagálnia az adott komponensnek. A szűrők három alapvető részből épülnek fel: ezek az action, a category, valamint a data. Az action, ahogy a neve is mutatja, meghatározza, hogy mit kell végrehajtania a meghívott komponensnek (illetve Broadcast Receivereknél magát a broadcast-et, a speciális eseményt mutatja). Számos ilyen akció van definiálva az Intent osztályban, például ACTION_CALL, mely egy híváskezdeményezést foglal magába, vagy az ACTION_MAIN, melyet az operációs rendszer az indításkor küld az alkalmazásnak. Míg az action döntően meghatározza a célalkalmazás viselkedését, a category opcióval lazább kapcsolatot, járulékos eljárásokat írhatunk le. Ilyen például a CATEGORY_LAUNCHER, mely az alkalmazások indítóképernyőjén való megjelenést biztosítja, vagy a CATEGORY_BROWSABLE, amely engedélyezi, hogy egy böngészőből is biztonságosan megnyitható az adott komponens, például egy e-mail elolvasására. A harmadik fontos rész, a data segítségével különböző feldolgozandó adatokat (Uniform Resource Identifier segítségével), és azok típusát adhatjuk meg, tipikusan a hozzájuk tartozó MIME-type-pal. Ez azért lényeges, mert egy audiolejátszó alkalmazásnak nem érdemes szöveges fájlt küldeni, hiszen annak feldolgozására nem alkalmas.

Ezt a három részt tehát a fejlesztő be tudja állítani a Manifest fájlban, mindháromból akár több szűrőt is definiálhat, valamint a küldendő Intent létrehozásánál, a megfelelő metódusok segítségével ez a három rész beállítható. Ezt követően implicit Intentet egy alkalmazás a szűrői alapján fogadhat. Amennyiben az adott szűrési feltételeknek az ér-

kezett Intent eleget tesz, megkaphatja azt az applikáció, egyéb esetben viszont blokkolva lesz az Intent. A szűrési feltételek különböznek az egyes szűrési komponensek között. Egy Intent objektum egyetlen action-nel rendelkezhet. Amennyiben ez az action benne van a célalkalmazás Intent-filterében, akkor ez a teszt sikeres volt, egyéb esetben elutasításra kerül az Intent. A következő a category tesztje: egy Intent objektum nem csak egy, hanem több kategóriával rendelkezhet, és a szűrés során mindegyik ilyen kategóriát definiálnia kell a célapplikációnak is az Intent szűrői között. Tehát egy kategória sem hiányozhat, amelyet az Intent tartalmaz, azonban több lehet a Manifest fájlban: ez azt is jelenti, hogy ha az Intent nem tartalmaz kategóriát, automatikusan átmegy ezen a teszten. Az utolsó teszt az adatrészre vonatkozik, itt mind az adat, mind a típus szűrésre kerül: amennyiben tartalmaz adatot és/vagy típust, mindkettőnek (ha létezik) meg kell egyeznie az Intent-szűrőkben felsorolt egyik adattal/típussal. Ha sem adatot, sem adattípust nem tartalmaz az Intent, automatikusan átmegy a teszten [1].

Természetesen előfordulhat az a helyzet, hogy több alkalmazás szűrőin is sikeresen átmegy egy Intent. Ekkor vagy az operációs rendszernek kell döntenie az elindítandó alkalmazásról, ez történhet prioritásos alapon (tipikusan a Broadcast Receivereknél így működik), melyet az Intentben lehet beállítani. Azonban egy másik lehetőség, hogy rábízza a döntést a felhasználóra, aki a megfelelő programok listájából választhatja ki a célalkalmazást [1]. Fontos megemlítenünk, hogy az Intent filterek önmagukban nem jelentenek biztonsági megoldást: attól, hogy bizonyos komponensek rendelkeznek ilyen szűrőkkel, nem jelenti azt, hogy más módon nem lehet őket meghívni. Az Intent-filterekkel definiált komponensek kaphatnak mind implicit, mind explicit Intentet, ezáltal a szűrők azonnal megkerülhetőek, hiába definiáltuk azokat úgy, hogy például csak bizonyos típusú adatokat fogadhat az adott szegmens. A szűrők pontos bemutatása a belőlük származó biztonsági problémák megértése miatt szükséges [1].

Először tekintsük a tényleges applikációk közötti kommunikációt, majd más problémák felmerülését vizsgáljuk. Ahogy az explicit Intenttel, úgy az implicittel is elindíthatunk egy komponenst, ráadásul az Intent-filterekből származó esetleges komplikációk (például több applikáció közül kell választani) is megkerülhetőek. Ezt úgy tehetjük meg, hogy nem az Intent osztályban specifikált akcióval hozzuk létre az Intentet: magunk is készíthetünk ilyen action-öket, egyszerűen a nevük megadásával, és ha ez egy egyedi azonosító, akkor egyértelmű lesz az operációs rendszer számára, hogy melyik komponenst kell meghívni. Ezzel a módszerrel az explicit Intenttel történő kommunikációhoz nagyon hasonló működés valósítható meg. A gyakorlatban is kipróbált alkalmazás ugyanolyan funkcionalitással bír, mint az explicit Intentes megvalósítás, így láthatóan a biztonsági problémák itt is felmerülnek.

Azonban implicit esetben nem csak a tényleges adatcsere jelenthet problémát. Ebben az esetben ugyanis nem lehet garantálni, hogy az Intent azt az alkalmazást indítja el, amelyet a fejlesztő elképzelése szerint el kéne neki indítania. Egy megfelelő alkalmazás "elkaphatja" az Intentet, és ezáltal különböző adatokhoz juthat hozzá, sőt, az Intent hatására elvégzendő szolgáltatások elvégzését akadályozhatja meg. Mindez jelentős problémákhoz vezethet. Ez történhet például Broadcast Receiver segítségével: amennyiben egy applikáció másoknak szeretné jelezni egy esemény bekövetkeztét broadcast-tel, akkor egy olyan alkalmazás, amely Intent-filterében minden egyes broadcastot, category-t és data-t elfogad, megszerezheti ezt a jelzést, és ekkor az eredeti célpontok nem szereznek tudomást az esemény bekövetkeztéről. Sőt, a küldő alkalmazás sem fog erről tudni, hiszen visszaigazolást nem kap az esemény kezeléséről. Ezek alapján fejlesztőnek mérlegelnie kell, hogy milyen információkat küld broadcast segítségével, és érdemes explicit Intenteket használ-

nia megoldásként, azonban erre nyilván nincs mindig lehetőség.

Fontos még megemlítenünk az úgynevezett Activity és Service Hijacking-et [8]. Egy alkalmazás eltéríthet egy implicit Intentet úgy, hogy az ő Activity-jét vagy Service-ét indítja el az operációs rendszer, és nem az eredeti célalkalmazás komponensét. Ez több okból is problémát okozhat. Egyrészt az Intentben tárolt adatot eltulajdoníthatja, vagy a felhasználót megtévesztheti: amennyiben az eredetileg is hívni kívánt applikációhoz hasonló interfésszel rendelkezik, a felhasználó azt hiheti, hogy a jó alkalmazás indult el. Ekkor személyes adatait a rosszindulatú alkalmazás megszerezheti, ráadásul hamis megerősítést is adhat egy nem bekövetkezett eseményről: például banki tranzakció esetén az adatokat megszerezve azt hazudhatja, hogy a tranzakció gond nélkül lefolytatódott. Activity esetén probléma lehet, hogy amennyiben több komponens szűrőin is átjutott az Intent, akkor a felhasználónak kell választania, hogy mi induljon el. Azonban ekkor egy megtévesztő név, vagy funkcionalitás becsaphatja a felhasználót, és akár azt is beállíthatja, hogy alapértelmezetten a kárt okozó alkalmazás induljon el. Service esetén ilyen lehetőség nincs, több megfelelő Service esetén véletlenszerűen választ a rendszer egyet, azonban látható, hogy ekkor is megvan az esély adatok eltulajdonítására.

3.1.6. Összefoglalás

Az Intentek tehát alapvetően két nagy típussal rendelkeznek. A biztonsági architektúra alapjait képező biztonsági megoldások segítségükkel kijátszhatóak. A fejlesztő nem megfelelő magatartása esetén mindkét típus segítségével könnyedén lehet szenzitív adatokat kiszivárogtatni. Az alkalmazás telepítése után nincs ellenőrizve az applikációk közötti kommunikáció, így ez a folyamat a felhasználó tudta nélkül zajlik, aki biztonságban érezheti magát. Érdemes lenne az Intentek működését felülvizsgálni, futásidőben is monitorozni a viselkedésüket, és nem a fejlesztőre bízni az esetleges biztonsági hibák kihasználását; konkrét megoldási javaslatokat a dolgozat későbbi fejezetében mutatunk be.

3.2. A sharedUserId

3.2.1. Az sharedUserId működése

Az Intentek működésének megismerése után egy újabb olyan eszközzel ismerkedünk meg, amelyet az Android biztosít az SDK-n keresztül, ez pedig a sharedUserId. Segítségével alkalmazások korlátlanul férhetnek hozzá egymás fájljaihoz, erőforrásaihoz, ezért mindenképp meg kell vizsgálnunk a viselkedését, hiszen potenciálisan Inter-Process Communication valósítható meg vele. A sharedUserId megléte a Linux kernelen alapul. Ahogy a biztonsági architektúra megismerésénél láttuk, a Linux más operációs rendszerekhez képest speciális biztonsági megoldásokat választott. Az egyik ilyen, hogy az alkalmazások egy "sandbox"-ban futnak, csak a saját fájljaikat használhatják fel, el vannak egymástól szeparálva. Erre a Linux felhasználói azonosítókat használ (user ID), és ez az azonosító az Android operációs rendszeren minden egyes alkalmazás esetén különböző, egyedi identifikálásra szolgál.

Androidon is lehetőség van arra, hogy két applikáció közös felhasználói azonosítóval rendelkezzen, mindössze az alkalmazások Manifest fájljaiban kell megadni ezt a közös azonosítót (azonban mindez korlátozásokkal jár, ahogy ezt később látni fogjuk). Ennek az lesz a következménye, hogy egymás fájljaihoz, erőforrásaihoz korlátlanul hozzáférnek, valamint egyetlen, közös folyamatban (process) futnak. Ennek a működésnek egy feltétele van, méghozzá az, hogy a két applikáció közös digitális aláírással rendelkezzen. Azonban

ez esetünkben nem probléma, hiszen a fejlesztő egyszerűen létre tud hozni saját magának ilyen kulcsokat, és a két alkalmazást publikálás előtt ugyanazzal a kulccsal aláírhatja.

3.2.2. A sharedUserId-vel való kommunikáció gyakorlati megvalósítása

Az IPC megvalósításának alapötlete az volt, hogy mivel elérik egymás fájljait, így nincs szükség Intentre az adatátvitelhez, megoldhatják az egyes alkalmazások ezt a fájlokon keresztül. A megvalósított alkalmazások az Intenteknél látott funkcionalitással bírnak, mindössze az adatátvitel módjában különböznek: az egyik alkalmazás csak a GPS koordinátákhoz kér hozzáférést, amennyiben változnak ezek a koordináták, egy fájlba kimenti azokat. A másik alkalmazás internet-hozzáféréssel rendelkezik, ez ki tudja olvasni a fájlból a kapott adatokat, és feltölteni azokat. Azonban az alkalmazások telepítése során problémába ütköztem. Bár az első alkalmazás telepítésénél a felhasználónak valóban csak az egyik jogosultságot kell megadnia, azonban a második alkalmazásnál már mind a helymeghatározáshoz, mind a hálózati hozzáféréshez engedélyt kell biztosítani. Ennek az oka ott keresendő, hogy a sharedUserId használata esetén a két alkalmazást egynek tekinti az operációs rendszer, nem tesz közöttük különbséget, így nyilvánvalóan szükség van a telepítéshez az összes szükséges jogosultságra. Látható tehát, hogy ez a rendszer jól kidolgozott, ebből a szempontból biztonságos, így az általunk eltervezett, a felhasználó tudta nélküli adatgyűjtés és -továbbítás itt nem sikerült.

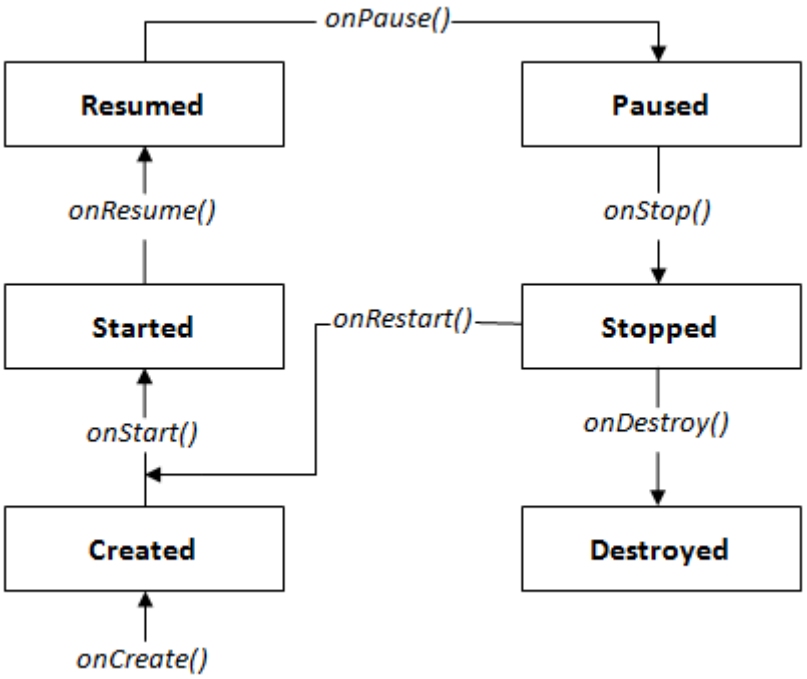
4. Alternatív megoldások vizsgálata

Az eddigi részekben láthattuk, hogy már az Android SDK is rendelkezik olyan fejlesztőeszközökkel, melyek segítségével alkalmazások közötti kommunikáció valósítható meg. Azt is demonstráltuk, hogy ezen eszközök nem megfelelő használata esetén szenzitív felhasználói adatok szivároghatnak ki. Azonban nem csak a fejlesztőrendszerben adott módszereket kell megvizsgálnunk, hanem alternatív, nem maguktól értetődő, eltérő implementációt igénylő megoldásokat is. A cél természetesen továbbra is az, hogy a rendszer kijátszásával, a felhasználó tudta nélkül juttassunk el adatot egyik alkalmazástól a másikig. Ezek egy része azon alapul, hogy az Android a Linux kernelre épül, így az ott megjelenő eszközök, megoldások ebben az operációs rendszerben is használhatóak.

A következő fejezetekben szó lesz a socketről, a signalról, és a kicsit kevésbé Linuxhoz kötődő, de annál lényegesebb fájlkezelési megoldásokról. Bemutatjuk az egyes eszközök működését, majd az ezek alapján felmerülő alapötletet, vagyis hogy miként lehetne Inter-Process Communicationt megvalósítani. Ezt követően ezek implementálása következik: azon kérdések megválaszolása, hogy megvalósítható-e az elképzelés, ha esetleg nem, akkor milyen további lehetőségeink vannak ezen az úton továbbhaladva, ha pedig igen, akkor milyen korlátokkal rendelkezik a megoldás. Így egy széleskörű, különböző területeken megvalósított megoldáshalmazt kapunk.

Lényeges az egyes megoldások függetlensége, tehát amennyiben lehetőség van rá, korábbi megoldásokat ne alkalmazzanak. Így ezekben az új megoldásokban kerüljük az Intentek használatát, bár látni fogjuk, hogy bizonyos esetekben nem lesz más lehetőségünk. Ezáltal sajnos egy erős eszköztől, a Service-tól esünk el, hiszen ezek kizárólag Intent segítségével indíthatóak el, és velük volt a legkönnyebb háttérben futó folyamatokat megvalósítani. Természetesen ettől még a két applikációnak egyszerre kell futnia, hogy egymással kommunikálni tudjanak. Ennek a megvalósításához adunk egy áttekintést az Activity-k életciklusáról, indulásuktól megszűnésükig [1]. Egy Activity életciklusának hét állomása van, melyek a 3. ábrán láthatóak. Láthatjuk, hogy az indítás után a Created, Started, Resumed állapotokon megy keresztül. Ezt követően, amennyiben az ablakot részlegesen eltakarja például egy szövegdoboz, vagy alvó állapotba kerül a telefon, Paused lesz az Activity. Amennyiben teljesen a háttérbe szorul a jelenleg aktív ablak (például a felhasználó elindít egy másik applikációt, vagy a jelenlegi alkalmazás egy másik aktivitását), akkor Stopped állapotba kerül. Innen az onRestart metóduson át Started, majd Resumed lehet ismét, ha a felhasználó visszatér az ablakhoz és az operációs rendszer még nem állította le; viszont a megszüntetése is lehetséges, ha a rendszer úgy dönt (például memóriahiány miatt), hogy teljesen leállítja az alkalmazást.

Az egyes implementációkban azt fogjuk kihasználni, hogy ha a kommunikáció egyik résztvevője a Stopped állapotban figyel az érkező információkra, akkor a másik a megfelelő módszer segítségével küldhet neki adatot. Természetesen számolni kell azzal az eshetőséggel, hogy a figyelő alkalmazást megszünteti az operációs rendszer: ennek a problémának a kivédése egyszerű forgalomszabályozási mechanizmusokkal megoldható, ez csak implementációs kérdés. A fő cél az alkalmazások közötti kommunikáció lehetőségeinek feltérképezése.



3. ábra. Az Activity életciklusa

4.1. A socketek

4.1.1. A socketek alapjai

Első alternatív megoldásként az úgynevezett socketek működését, használatát vizsgáljuk meg. A socket segítségével tulajdonképpen két végpont közötti hálózati kapcsolat létrehozására van lehetőség. Az egyik végpont egyik portját kinyitja a külvilág felé (létrehoz egy socketet a specifikált porttal), és az ezen a kapcsolaton keresztül érkező adatokat képes fogadni, feldolgozni, műveleteket végezni azokon. A socket tehát az egyik legegyszerűbb, legalapvetőbb eszköz adatok küldésére és fogadására hálózati kapcsolaton keresztül.

A Java API rendelkezik Socket osztállyal, amely segítségével pontosan ezt a funkcionalitást tudjuk megvalósítani. Programjainkból képesek vagyunk egy port megnyitására, és az azon bejövő adatok fogadására az osztály metódusainak használatával. Hasonlóan, egy adott eszköz címének és portjának megadásával megkísérelhetünk csatlakozni, és amennyiben ez sikerült, adatok küldhetünk a másik végpontnak. Mivel az Android SDK támogatja a Socketek használatát (ez azért fontos, mert bizonyos, Javában használható fejlesztőeszközöket nem feltétlenül használhatunk Androidon), ezt a működést Androidos készülékeinken is megvalósíthatjuk.

Eddig azonban két különböző eszköz kommunikációjáról volt szó, nekünk viszont nyilvánvalóan egyetlen rendszer áll rendelkezésre, ezen kell két applikációnak kommunikálnia. Mindez az úgynevezett loopback address segítségével valósulhat meg. Ennek a címnek a segítségével egyetlen, önálló rendszeren is hálózati tranzakciók szimulálhatóak: az erre a címre érkező kéréseket, adatokat az operációs rendszer úgy dolgozza fel, mintha külső forrásból származó információkról lenne szó, tehát tulajdonképpen egy virtuális hálózati interfészről, kapcsolatról van szó. Ezek alapján adja magát az alapötlet: amennyiben az applikációink tudnak ezen a címen portot nyitni socket segítségével, és erre adatot is

képesek küldeni, akkor bármilyen adatot megoszthatnak egymással.

4.1.2. Megvalósítás a gyakorlatban

Az implementáció kiindulópontjában az alkalmazások az eddig megszokott jogosultságokkal rendelkeznek: az egyik a helykoordinátákhoz, a másik a hálózati kapcsolathoz kap engedélyt. Mindkét alkalmazás socketet használ a kapcsolat megvalósítására, a fogadó nyit egy tetszőleges portot a visszacsatolási címen, és ezt követően ott figyel. Ahogy azt az előző fejezetben leírtuk, az egyik alkalmazásnak Stopped állapotban kell részt vennie a kommunikációban az Intent elkerülésének érdekében, esetünkben ezt a pozíciót a fogadó alkalmazás tejesíti. Amint a portot kinyitotta, a másik applikáció szintén socket segítségével csatlakozhat a loopback címre, és adatokat küldhet egy adatfolyam segítségével, melyre szintén a Socket osztály ad lehetőséget. Ezeket az adatokat a fogadó fél megkapja és fel tudja őket dolgozni.

A programok első kipróbálásakor azonban egy jelentős problémába ütköztem: az operációs rendszeren bármilyen hálózati kapcsolat használatához, legyen az egy Wi-Fi kapcsolat, vagy egyszerű socketek használata, jogosultságra van szükség. Sajnos, bár esetünkben valós hálózati kommunikáció nem valósul meg, mégis szükséges a Manifest fájlban a megfelelő jogosultságok megadása. Mindez viszont azt jelenti, hogy telepítéskor mindkét applikációnak engedélyt kell adni a hálózat használatára.

Ez alapvetően rányomja a bélyegét a megoldásra, azonban a biztonság kivitelezése kifogásolható marad: a kommunikáció két alkalmazás között a loopback address-en történik, így tényleges hálózati kapcsolatra nincs szükség. Amennyiben a felhasználó a telepítés után internetkapcsolat nélkül használ két ilyen kommunikációra képes alkalmazást, vagy a hálózat bekapcsolása előtt leállítja a forrásapplikációt, biztonságban érezheti magát, hiszen azt hihetné, hálózati kapcsolat nélkül nem fognak kárt tenni ezek az alkalmazások. Azonban több alkalmazás ilyen formán történő közös kommunikációja továbbra is megvalósulhat, így a felhasználó félrevezetésével cserélhetnek offline módon is adatot az applikációk, mely a biztonsági rendszer gyengeségének tudható be.

4.2. A signal

4.2.1. A signalok alapjai

A signalok a UNIX rendszereken speciális események jelzésére szolgáló eszközök. Az egyes alkalmazások (illetve folyamatok, szálak) ezeket a jelzéseket fogadhatják, és feldolgozhatják: a signal típusától függően meghívódik a megfelelő kezelő eljárás, és elvégzi a kívánt műveleteket reakcióként. A signalok működése aszinkron módon történik, így egy ilyen jelzés érkezésekor a program végrehajtása megszakad, és a signal kezelése után folytatódik. Amennyiben az adott signalhoz a fogadó fél rendelkezik hozzárendelt signal handler-rel (signal kezelővel), akkor ez fog lefutni, egyéb esetben az alapértelmezett kezelő eljárásé lesz a signal feldolgozásának feladata.

A UNIX alapú rendszerek, így a Linux is többféle signallal rendelkezik: ilyenek többek között a SIGQUIT, amelyet a felhasználó által kért leállítás esetén kap meg a folyamat, vagy a SIGALRM, mely egy időzítő lejártakor jelez az alkalmazásnak. Két megkülönböztetett signallal is rendelkeznek ezek a rendszerek: ezek a SIGUSR1 és SIGUSR2, melyek a felhasználó által definiált jelzések. Mindez választ ad arra a kérdésre, hogy miképp szeretnénk a signalokkal alkalmazások közötti kommunikációt megvalósítani: két különböző jelzés segítségével (például a USR1 és USR2) bináris adatot küldhetnénk egyik applikációból

a másikba. A signalokat egy nagyon egyszerű, általunk megvalósított handler feldolgozná, és az adatátvitel a háttérben, a felhasználó számára észrevétlenül történhetne.

4.2.2. Signal küldése UNIX rendszeren

Signalokat különböző események automatikusan kiválthatnak, például a korábban is bemutatott időzítő lejárása, vagy egy olyan memóriaterületre való hivatkozás, melyhez az adott alkalmazásnak nincs jogosultsága. Maga a felhasználó is küldhet signalt egy-egy futó folyamatnak: a terminálban a kill parancs segítségével, a folyamat azonosítójának és a signal nevének megadásával tudunk jelzéseket küldeni. Az említett azonosítót a szintén a terminálban használható ps parancs segítségével tudhatjuk meg: ez kilistázza a futó folyamatok neveit, egyedi azonosítóit és egyéb információit, beállítástól függően. Mindez a kommunikáció lehetőségének feltérképezésénél fontos lesz számunkra.

4.2.3. Signal küldése Java alkalmazásból

Természetesen esetünkben az IPC megvalósításához nem a felhasználónak, hanem az egyes alkalmazásoknak kell signalt küldeniük. Ahogy láthattuk, az első lépés a megfelelő folyamatazonosító meghatározása, amelynek a jelzést küldeni szeretnénk. Ezt a Java segítségével megtehetjük, méghozzá nagyon hasonlóan, ahogy egy felhasználó is megteszi azt: a ps parancs kimenetéből kikereshetjük a nekünk megfelelő alkalmazást. Mivel a célalkalmazás nevét tudjuk, hiszen azt a fejlesztő határozza meg, a hozzá tartozó azonosítót kinyerhetjük a következő módon: a Java segítségével folyamatok indíthatóak el, méghozzá a Runtime.getRuntime().exec() metódussal, ahol az exec() paramétere az elindítani kívánt folyamat neve. Mindez egy Process osztálybeli objektummal tér vissza. Tehát ha a ps-t lefuttatjuk ezzel a módszerrel, akkor megkapjuk a hozzá tartozó futó folyamatot. Azonban ekkor az ő által (alapesetben képernyőre) kiírt kimenetéből tudjuk csak meghatározni a szükséges információt.

Ehhez is a Javát hívhatjuk segítségül: a Process osztály rendelkezik többek között getOutputStream() metódussal, melynek segítségével az adott folyamat kimeneti adatfolyamát kaphatjuk meg. Ez a ps esetén az összes futó folyamatot és azok azonosítóját tartalmazza. Tehát ennek egyszerű beolvasásával és a megfelelő folyamatnévhez tartozó azonosító megkeresésével a célalkalmazás identifikálható. Ezt követően a signalt kell valamilyen módon eljuttatni hozzá.

A Java rendelkezik olyan csomagokkal, melyekkel a signalok küldése és kezelése nagyon leegyszerűsödik: a sun.misc csomag Signal és SignalHandler osztályaival lehet mindezt elvégezni. Azonban az Android SDK ezt nem támogatja, így más utat kellett találnunk a signalok küldésére és fogadására. Signalt hasonlóan a ps futtatásához, elvileg kill parancs segítségével ugyan úgy tudunk küldeni, sőt az API-ban is található erre eszköz: az android.os.process csomag Process osztályának sendSignal() metódusa alkalmas erre, azonban különös módon ehhez tartozó kezelő eljárás nincs. Ezért egy új utat kellett találni a jelzések fogadására és kezelésére, ez pedig az Android natív kódú programozása lett.

4.2.4. Az Android NDK

Eddig az alkalmazásaink az Android SDK segítségével, Java nyelven íródtak. Viszont lehetőség van natív, C illetve C++ kód futtatására is az Androidos eszközeinken. Mindezt az Android Native Development Kit (NDK) teszi lehetővé. Segítségével bizonyos egyszerű

formai követelményeknek megfelelő, natív forrásfájljainkat egy SDK-s applikációhoz hozzáfűzhetjük, és az így definiált függvények, eljárások már futtathatóak lesznek a készülékeinken. Mivel maga a Linux is tulajdonképpen teljes egészében C-ben íródott, ezáltal az így kreált alkalmazások sokkal közelebb állnak a rendszermaghoz, így újabb programozói eszközök használatához nyílik út számunkra.

Esetünkben tehát azért fontos az NDK használata, mert így az olyan, a Linux alapjait képező függvényekhez, rendszerváltozókhoz, és egyéb fejlesztői eszközökhöz kapunk hozzáférést, amelyek használatára Java alatt nem lenne lehetőségünk. Ilyen eszköz például maga a korábbról ismert kill() parancs, a signalok kezelésére vonatkozó függvények, vagy különböző hibaindikátor változók, melyek szerves részét képezik a vizsgálat felépítésének.

4.2.5. A signalok kezelése natív kód segítségével

Amennyiben signalokat kap egy program, ezek egy várakozási sorba kerülnek, és itt is maradnak, amíg fel nem dolgozzák őket. A signalok feldolgozására különböző metódusok állnak rendelkezésünkre a Linuxos alapoknak köszönhetően. Talán a legfontosabb közülük a sigwait(). A sigwait első paramétere egy sigset_t struktúra. Egy ilyen halmazban minden egyes signaltípusról el lehet dönteni, hogy eleme-e ennek a halmaznak, vagy sem. Az elemeket különböző függvények segítségével tudjuk beállítani minden set esetén. A sigwaitnél ez a halmaz azt mondja meg, hogy mely signalok érkezése esetén fusson le a függvény. Amennyiben a sigwait() hívásakor a rendszer várakozási sorában a legelső signal-t képes fogadni a sigwait (tehát benne van a neki paraméterül átadott set-ben), akkor a sigwait kiveszi a signalt a sorból, lefut a függvény, visszatérési értéke pedig a kapott signal típusa lesz. Célunk az volt, hogy a sigwait()-et követően a kapott signal függvényében különböző metódusok futhatnak le, így minden egyes általunk kezelni kívánt jelzéshez saját signal handler-t írhatunk.

4.2.6. Gyakorlati megvalósítás

Minden Java alkalmazást egy virtuális gép futtat, nincs ez másképp az Android esetében sem. Ahogy korábban is láthattuk, ezt a virtuális gépet Dalvik VM-nek (Dalvik Virtual Machine) nevezik az Android esetén. A Dalvik VM forráskódjából megbizonyosodhatunk arról, hogy alapértelmezetten a signalok kezelését ő végzi el, ráadásul nincs is olyan gazdag rendszere a signaloknak, mint azt a UNIX-os rendszereken láthattuk: a forráskód alapján mindössze 3 signal, a QUIT, a USR1, és a USR2 kerül kezelésre. Ez a megvalósítás szempontjából kedvező, hiszen az általunk használni kívánt signalok pont rendelkezésre állnak, illetve azt is tudjuk, hogy más signallal valószínűleg nem érdemes próbálkozni, hiszen kezelésük nem valósul meg.

Ennek tudatában kezdődött meg a fent bemutatott megoldás implementálása. A signal handler elkészült, a megfelelő signalok fogadására alkalmassá lett téve (tehát a paraméterül átadott set-ben a három signal be lett állítva). A natív kódot is tartalmazó fogadó alkalmazás Stopped állapotban várta a signalokat. A szintén korábban bemutatott módszerrel könnyedén meg lehet határozni az azonosítóját, és először a sendSignal metódussal próbálkoztam, sikertelenül. A fogadó applikáció nem lépett ki a sigwait()-ből, tehát nem kapott megfelelő jelzést. Ezt követően ki lett próbálva a rendszer ugyanebben a felállásban, de más signalküldési mechanizmussal: a Runtime.getRuntime().exec() metódussal a kill parancsot futtattam a megfelelő paraméterekkel, hiába. Természetesen mindhárom engedélyezett signal küldését kipróbáltam ilyen formán, sőt, a Dalvik VM által nem elkapott signalokat is (nyilvánvalóan ekkor a sigwait()-nek átadott setben ezek a signalok is

engedélyezve voltak). Úgy tűnt, egyszer sem sikerül eljuttatni a fogadóhoz a signalt, így más módszerhez kellett folyamodnom.

Először arra gondoltam, hogy jó lenne még a sigwait előtt eljuttatni a signalt az alkalmazáshoz, mindehhez pedig a natív raise() függvényt alkalmaztam. A raise() segítségével egy alkalmazás önmagának küldhet signalt, tulajdonképpen a kill(getpid(), signal) függvény rövidített változata, ahol a getpid() a folyamat azonosítóját adja vissza. Meghívtam tehát a sigwait() előtt, és ekkor valóban lefutott a sigwait(), megtörtént a megfelelő signal kezelése. Sajnos ekkor a siker okozójának azt a tényt könyveltem el, hogy sigwait() előtt kell kapnia signalt az alkalmazásnak, így immár nem a raise() segítségével, hanem sendSignal()-lal próbáltam ugyanezt megvalósítani. Mindez úgy történt, hogy a fogadó alkalmazásban a sigwait() csak az onRestart() metódusban indult el: emlékeztetőül, ez akkor fut le, ha az alkalmazást elhagyja a felhasználó, majd visszatér oda (nyilvánvalóan erre a valóságban nem lehet építeni, azonban a tesztelés érdekében jó megoldásnak tűnt). Így a fogadó alkalmazást elindítva, majd a küldőből (ekkor Stopped lesz a fogadó) signalt küldve és a fogadóba visszalépve ugyanazt azt eredményt kellene kapnunk, mint raise() segítségével. Azonban sikertelen volt a próbálkozás. Ugyanezzel az eredménnyel járt a kill-el való signal küldése ezzel a módszerrel, így egészen erős lett az a sejtésem, hogy nincs lehetőség egyik applikációból a másikba signalküldésre. Ezt azonban be kell bizonyítani.

Szükség van tehát egy olyan eszközre, amely valamilyen visszajelzést ad egy signal küldésének sikerességéről, erre pedig képes a natív kill() függvény. Visszatérési értéke -1, ha valamilyen problémába ütközött, azonban ami még fontosabb: a Linux rendelkezik egy errno nevű rendszerváltozóval, amelyet bizonyos függvények képesek beállítani, ha valamilyen hiba történt a végrehajtásuk során. Erre a kill() is képes, és -1-es visszatérési értékénél három lehetséges értéke lehet a hibajelző változónak: `EINVAL` (a megadott signal érvénytelen, vagy nem támogatott), `EPERM` (nincs jogosultság a megadott folyamatnak signal küldésére), valamint `ESRCH` (nem található a megadott azonosítójú folyamat). A natív kill() sem tudta elküldeni a signalt természetesen, és lefutása után az errno változó `EPERM` értékű lett. Ez alapvetően egy jó jel, hiszen a sejtés igazolódni látszik, azonban azt is meg tudjuk mutatni, hogy ez csak különböző alkalmazásoknál lép fel, és ezek a signalkezelő osztályok és függvények nem feleslegesen vannak az operációs rendszerben: a korábban bemutatott sharedUserId segítségével közös felhasználói azonosítót adtam a két alkalmazásnak. Ekkor mind sendSignal, mind kill segítségével sikeresen tudtam signalt küldeni. Ezek alapján azt a következtetést vonhatjuk le, hogy Androidon különböző azonosítójú alkalmazások egymásnak nem képesek signalt küldeni.

4.3. Kommunikáció folyamatok segítségével

4.3.1. A ps parancs

Bár signalok kommunikációra való felhasználására a jelek szerint nincs lehetőségünk, az elért eredményeket hasznosíthatjuk. Signal küldéséhez meg kellett határoznunk a célfolyamat azonosítóját, mindezt a ps parancs segítségével tehettük meg. Ezt természetesen minden Androidos applikáció használhatja, speciális jogosultság a használatához nem szükséges. Mivel esetünkben a kommunikáció mindkét oldalán álló alkalmazás ki tudja ezt a listát olvasni, esetlegesen adatot is át lehetne itt vinni. Az ötlet tehát következő: a küldő fél folyamatokat indít el és állít le periodikusan, míg a fogadó figyeli a ps-t, és ha megfelelő folyamatnevet észlel, akkor tudja, hogy információ érkezett. Tehát ha adott egy A (küldő) és B (fogadó) alkalmazás, akkor ha az A elindítja a 0-ás bitet jelentő C folyamatot, vagy az 1-es bitet jelentő D folyamatot, akkor a B minderről értesülhet és

eltárolhatja a kapott adatokat; minderre lehetőséget a ps parancs biztosít.

4.3.2. Az implementáció

A gyakorlati megvalósítás első látásra egyszerűnek tűnik: a küldő alkalmazásnak csak folyamatokat (processeket) kell indítania és leállítani, az adatok fogadása pedig egy másik applikáció feladata. Bár mind a Java API, mind az SDK rendelkezik Process és ProcessBuilder osztályokkal, ezekkel rendszerfolyamatok elindítására van lehetőségünk, azonban nekünk olyan processre van szükség, melynek nevében adatot tudunk elhelyezni, és tulajdonképpen nem csinál semmit, egyszerűen elindul, majd leáll. Mindezt Androidon Service-ok segítségével tudjuk megtenni. Ez azt jelenti, hogy Intenteket kell használnunk, azonban itt teljesen más minőségükben, mint korábban. Itt ugyanis az Intent hibáit tulajdonképpen nem használjuk ki: a küldő alkalmazás a saját szolgáltatásait indítja el, más alkalmazásét soha; az Intentek soha nem rendelkeznek extra, hozzájuk csatolt adattal; végül az Intent által elindított szolgáltatás nem csinál semmit: nem létesít hálózati kapcsolatot, nem indít el új komponenseket, így gyanús kóddal nem rendelkeznek. Ez azt jelenti, hogy az Intentek korábban látott gyengeségeinek kijavítása, ellenőrzése esetén is működőképes lehet ez a megoldás, ezért döntöttem ezen eszközök használata mellett.

A Service-ok segítségével tehát a következőképp lehet a megfelelő működést megvalósítani: a Manifest fájlban minden egyes komponensnek meg lehet adni, hogy amennyiben azt szeretnénk, hogy külön process-ben fusson, akkor milyen folyamatnevet adjon neki a rendszer. Első lépésként két ilyen Service volt, melyek a 0 és 1 értékeket reprezentálták ilyen formán. A folyamatok leállításánál először kisebb problémába ütköztem, hiszen alapvetően az operációs rendszer dönt egy-egy ilyen process leállításáról, ezt például akkor teheti meg, ha erőforrások felszabadítására van szükség. Ezt a problémát sikerült azzal megoldani, hogy a Java-ból ismert System.exit() metódus leállítja az adott alkalmazást futtató virtuális gépet, ezáltal megszűnik a folyamat, és ez a gyakorlatban valóban a megfelelő működést produkálta: a Service indítása után a ps-ből kikereshető volt, az exit() lefutása után viszont megszűnt, a listában sem jelent meg. Természetesen a hozzá tartozó applikáció továbbra is fut, mindössze a Service folyamata állt le. Ezzel módszerrel tehát képesek vagyunk bináris adatok küldésére, viszont ezek megfelelő fogadását is biztosítani kell.

4.3.3. Adatátvitel feltétel nélküli forgalomszabályozással

A megvalósítást először feltétel nélküli forgalomszabályozással próbáltam megvalósítani. Ez azt jelenti, hogy sem a fogadó fél nem értesül arról, hogy érvényes adat érkezett, sem a küldő nem kap jelzést, ha az érvényes adatot megkapták, és küldheti a következőt.

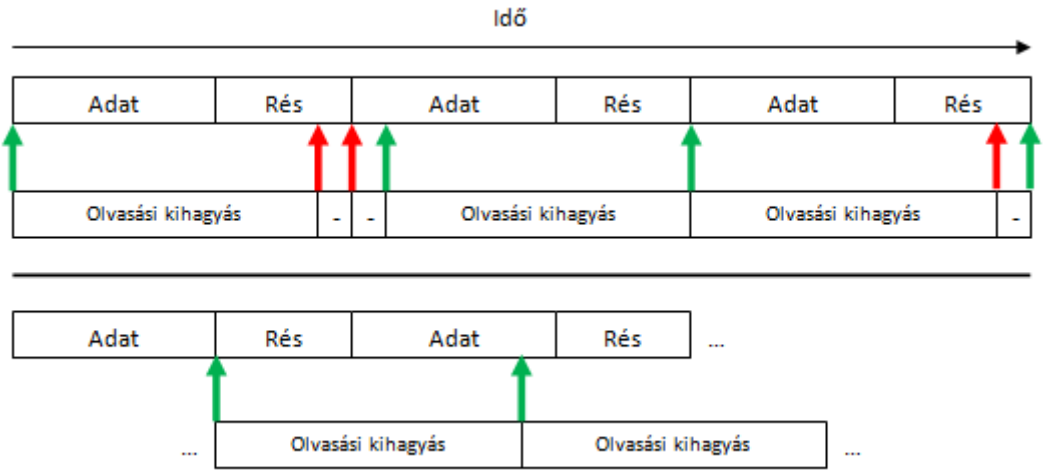
A hálózati kommunikációban jól ismert Manchester-kódolás segítségével valósítottam meg az adatátvitelt: ennél a kódolási sémánál a 0 bit-et egy alacsony jelszintből magasba (0-ból 1-be), míg az 1 bit-et magasból alacsonyba (1-ből 0-ba) váltással reprezentálunk (előfordulhat, hogy fordítva rendelik hozzá a bitekhez a változásokat, de ennek nincs jelentősége). A Manchester-kódban küldött adatok fogadására és dekódolására nincs szükség órajelre, mindez megoldható olyan formán, hogy az érkező adatfolyamot sűrűn mintavételezzük (sűrűbben, mint ahogy az egyes biteket a küldő kiadja), és a kapott sorozatból egy egyszerű algoritmus segítségével dekódolható az eredeti üzenet. Mindez sikeresen meg is valósult, azonban nem feltétlenül ideális sebességgel: az első tesztek során kifejezetten lassú működésre törekedtem a hibák felderítésének érdekében, ezért egy folyamat (ami az adatfolyamban egy jelszintet reprezentál, így egy jelszintváltozáshoz két folyamat egymás

utáni futására van szükség, ez fog 1 bitet jelenteni az üzenetből) 4,5 másodpercig futott, míg körülbelül kétszer ilyen gyorsan olvasott a fogadó fél. Néhány, binárisan kódolt decimális szám átvitele így sikerrel lezajlott, azonban egyértelmű az adatátvitel lassúsága, amely valós működési körülmények között nem megfelelő. Gyorsítani próbáltam az átvitelt, azonban minél gyorsabbá vált, egyre több hiba csúszott az adatátvitelbe: bithibák fordultak elő, esetlegesen ki is maradtak bitek, így nyilvánvaló volt, hogy ennél egy gyorsabb és megbízhatóbb megoldást kell találnunk.

A sebességet a szimbólumok számának növelésével tehetjük meg: az első lépés, hogy nem csak 0-1 biteket, hanem 0-tól 9-ig egész számokat reprezentáló folyamatokat indítunk el. Ezzel egy decimális szám átviteléhez nincs szükség 4 bitre, hanem egyetlen folyamattal képesek vagyunk a küldésre. Amennyiben lehetőségünk lenne a folyamatok nevének megválasztására futásidőben, akkor gyakorlatilag sokkal több adatot képesek lennénk átvinni, hiszen a folyamat neve hordozhatná ezeket az információkat, melyeket dinamikusan hozunk létre; itt azonban csak a Manifest fájlban tudjuk ezt a nevet megadni, így nincs erre lehetőség futás közben. A másik sebességnövelő tényező az lett, hogy az átvitel során olyan módon választjuk meg a küldési és olvasási időket, hogy biztosan minden küldött adatot beolvassunk, de ne olvassunk be egy adatot kétszer. Mindez sajnos sok munkával járt, hiszen meg kell találnunk az ideális időzítéseket, és ezeket le is kell tesztelnünk, hogy biztosan jó működést érjünk el.

A kommunikáció folyamata tehát a következő: a küldő fél vár a fogadóra, amikor észleli, hogy jelen van és az adatok fogadására kész, megkezdődik az adatátvitel. Esetünkben decimális számok küldését valósítjuk meg, hiszen ezen megfelelően prezentálhatjuk a módszer működését, és nyilván akár karakterek is küldhetőek lennének ezen a csatornán. A megfelelő decimális számhoz tartozó Service-t elindítja a küldő applikáció, eközben meghatározott időközönként figyel a fogadó alkalmazás, hogy érkezett-e már adat. Amennyiben igen, beolvassa azt, és vár a következőre. Nyilvánvalóan nem olvashatja ugyanazt azt adatot kétszer, ezért az olvasási periódust és a Service futási idejét megfelelően kell megválasztanunk. Azt is biztosítanunk kell, hogy a fogadó egymás után be tudja olvasni az érkező adatokat, hiszen nem ad értesítést a küldőnek a beolvasásról, ezért a küldő alkalmazás folyamatosan indítja a megfelelő folyamatokat.

Mindez a következőképp valósul meg: a Service egy meghatározott ideig fut, ezt követően viszont egy viszonylag hosszabb időtartamot követően nem küldünk adatot. Ennek két oka van: a folyamatok leállása nem történik meg azonnal, kis időre van szükség, míg teljesen leállnak és eltűnnek a processek listájából. Másrészt ez egy biztonsági időrésként szolgál az olvasás segítésének érdekében. Amennyiben észlelünk egy adatot a fogadó oldalon, beolvassuk azt, azonban egy, a folyamat futási idejénél hosszabb ideig nem olvasunk innentől. Ez megakadályozza egy folyamat kétszeri beolvasását, azonban vigyáznunk kell, hogy egy másik adatot nehogy átugorjunk, és ebben segít minket biztonsági időrés: a folyamat futási ideje plusz a biztonsági időrés már nagyobb mint az olvasási idő "kihagyása", így a következő adatot biztosan be tudjuk olvasni. A jobb érthetőség érdekében tekintsük a 4. ábrát, amely ezt a működést mutatja be. Az ábrán a zöld nyíl az adat sikeres olvasását jelenti, míg piros esetén ez nem valósul meg (például nem indult el a folyamat). Az ábra felső részén az adatátvitel általános sémáját láthatjuk. A csúszás a második adatnál nem számít, hiszen nagy biztonsággal tudjuk továbbra is beolvasni az adatot, ráadásul a következő ciklusban akár az elsőhöz hasonlóan, az adat közvetlen kiadásakor olvashatunk, tehát még a korrigálást is elvégzi ez az üzemmód. Láthatjuk, hogy az olvasási kihagyási idő után új adatra várunk, ezt viszonylag kis időnként ellenőrizzük a minél kisebb csúszás érdekében. Tulajdonképpen adat kihagyása nem lehetséges, hiszen az időzítések megfelelő



4. ábra. Adatátvitel folyamatokkal forgalomszabályozás nélkül

specifikálása esetén az adat kiadása plusz az időrés nagyobb, mint ez az olvasási kihagyás. Ezt láthatjuk az ábra alsó részén: amennyiben valamilyen okból csak egy adatrész legvégén tudunk beolvasni (erre viszont nagyon kicsi az esély), még ekkor sem ugrunk át adatot a nagy olvasási kihagyási idővel, és a korrigálás miatt a következő adatot már nem a legvégén, hanem kicsivel előtte is be tudjuk olvasni. Egy adat kétszeres beolvasását pedig a megfelelően nagyra választott "üresjárat" akadályozza meg.

A tesztelés feladata ezen időadatok meghatározása volt olyan módon, hogy nagyobb mennyiségű adat egymás utáni átvitele is biztosan jól működjön. Az egyes időzítésekkel először tíz, majd száz szám átvitelét teszteltem le, illetve a végső, legjobb megoldásnak bizonyuló időzítésnél ezer szám átvitelét is kipróbáltam, azonban ez csak a megoldás teljesítményét demonstrálja, valós helyzetben aligha van szükség ennyi adat közvetlenül egymás utáni átvitelére.

Néhány sikertelen próbálkozás után sikerült egy elsőre jónak tűnő időzítést választani: 800 ms-ig futnak az adatot reprezentáló folyamatok, 200 ms az időrés, és 1000 ms-onként végezzük el az olvasást. Láthatjuk, hogy itt két oldal időzítése megegyezik (mindkettő egy másodperc), ez még nem rendelkezik azzal a tulajdonsággal, hogy csúszás esetén képes korrigálni önmagát. Tíz szám esetén ez még nem volt probléma, több száz teszteseten keresztül hiba nélküli volt az átvitel. Azonban 100 számnál már megtörtént az elcsúszás, így ez a megoldás nem tökéletes, tovább kellett próbálkoznom. Az olvasási kihagyást lecsökkentettem 900 ms-ra, ezáltal az elcsúszás javíthatóvá válna, viszont újabb problémába ütköztem. Száz számon végeztem innentől fogva a teszteket, hiszen ez már releváns eredményt ad, és a tesztesetek kimenetein azt tapasztaltam, hogy bizonyos számok kétszer szerepelnek, míg azok, amelyek utánuk következnének, egyszer sem.

A probléma oka a következő: egymás után következő számokon végeztem ezeket a teszteket, ps a futó processeket azonosító szerinti növekvő sorrendben listázza ki, így amennyiben két adatküldő folyamat is fut, mindig a kisebb számot tartalmazó lesz elől, hiszen az a folyamat indult el előbb (és annak nem történt meg az időbeni leállása). Mivel a fogadó alkalmazás az első találatra reagál, beolvassa azt adatként. Tehát amennyiben egy process túl lassan áll le, akkor előfordulhat, hogy a sorban következő, érvényes adatot hordozó process-szel együtt lesz jelen a ps-ben, így kétszer olvassa be a fogadó alkalmazás, míg az érvényes adatot egyszer sem.

	10 szám		100 szám	
Adat-Rés-Kihagyás (ms)	Átlagos hiba	Hibavalószínűség	Átlagos hiba	Hibavalószínűség
800-200-900	0	0%	15	57%
700-300-900	0	0%	1	6,6%
600-400-900	0	0%	0	0%

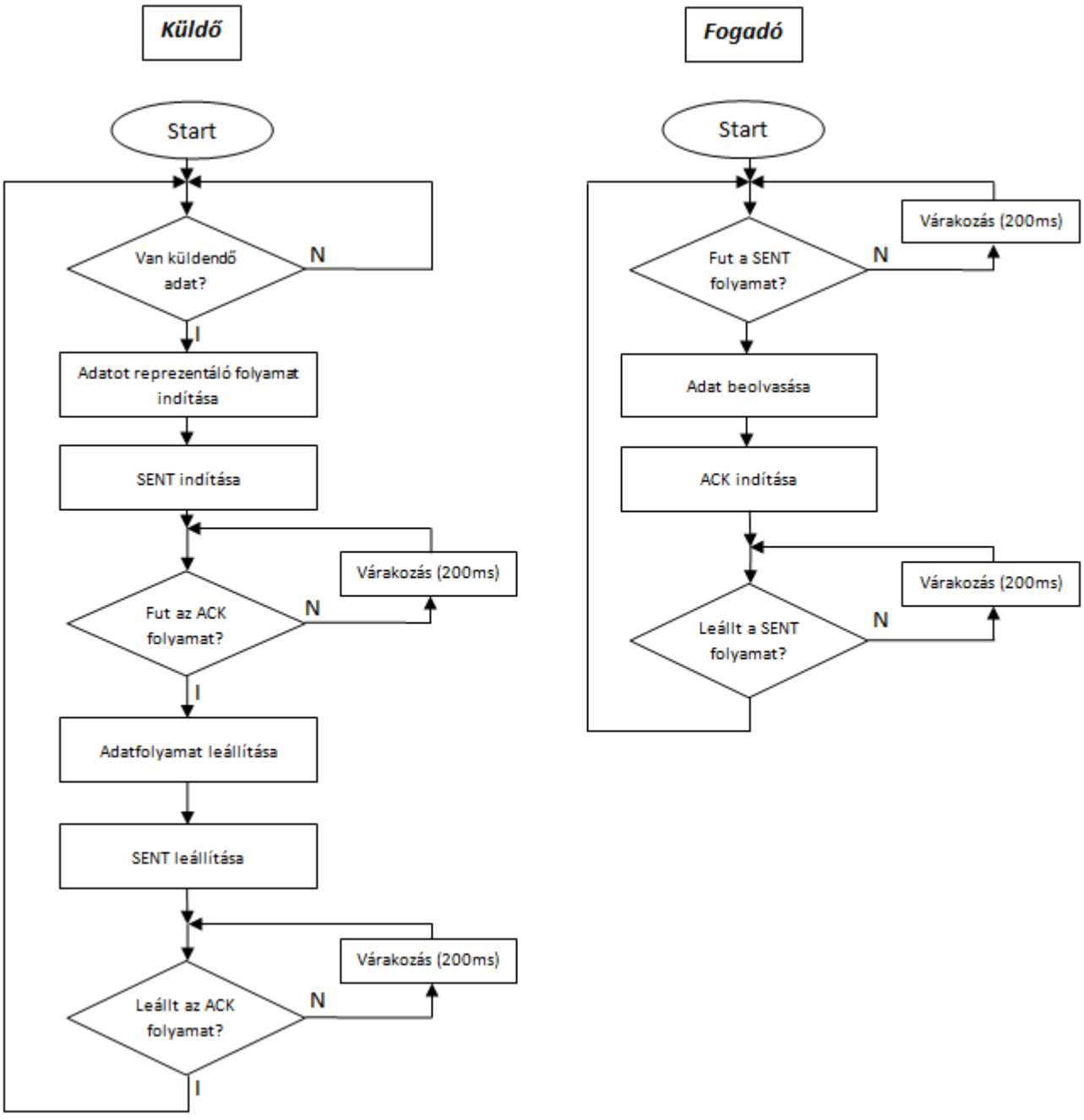
2. táblázat. Néhány adatátviteli eredmény a különböző időzítésekkel

Ahogy fentebb is írtam, a biztonsági résnek kell megfelelő nagyságúnak lennie ennek megakadályozásához. Tovább próbálkoztam az eddigi 200 helyett 300 ms-mal, kevesebb hibát kaptam, de még mindig nem volt tökéletes, az egyes adatátviteli eredmények a 2. táblázatban láthatóak. Megnöveltem a rést 400 ms-ra, ekkor a küldő folyamat 600 ms-ig fut, az olvasás maradt 900 ms-onként. Száz szám átvitelét mintegy 450-szer teszteltem, egyszer sem történt hiba az átvitel során, a fogadó alkalmazás mindig megkapta a küldő által kiadott adatokat. Sőt, ezer szám kipróbálását is elvégeztem, sikeres volt ez a teszt is. A megvalósítás megbízhatóságával tehát viszonylag kis mennyiségű szám esetén biztosan nem lesz probléma. Az adatátviteli sebesség is megfelelő: az időzítési adatokból látható, hogy 1 decimális szám átvitele kb. 1 másodpercet vesz igénybe. A szélességi és hosszúsági koordináták 8-8 számjegy esetén már nagyon pontosak, ennek a 16 számnak az átvitele mintegy 16 másodpercbe kerül. Ez a megoldás tehát úgy tűnik, nagy biztonsággal működik, ráadásul jóval nagyobb sebességgel, mint az előző: földrajzi koordináták átviteléhez nincs szükség percekre, másodpercek alatt megtörténhet, mely a valós körülmények között teljesen megfelelő. A tapasztalat azt mutatja, hogy ennél lényegesen gyorsabb eredmény nem valósítható meg, a folyamatok meglehetősen lassú indulásának és leállásának következtében, mindezt láthatjuk a következő, forgalomszabályozással megvalósított részben is.

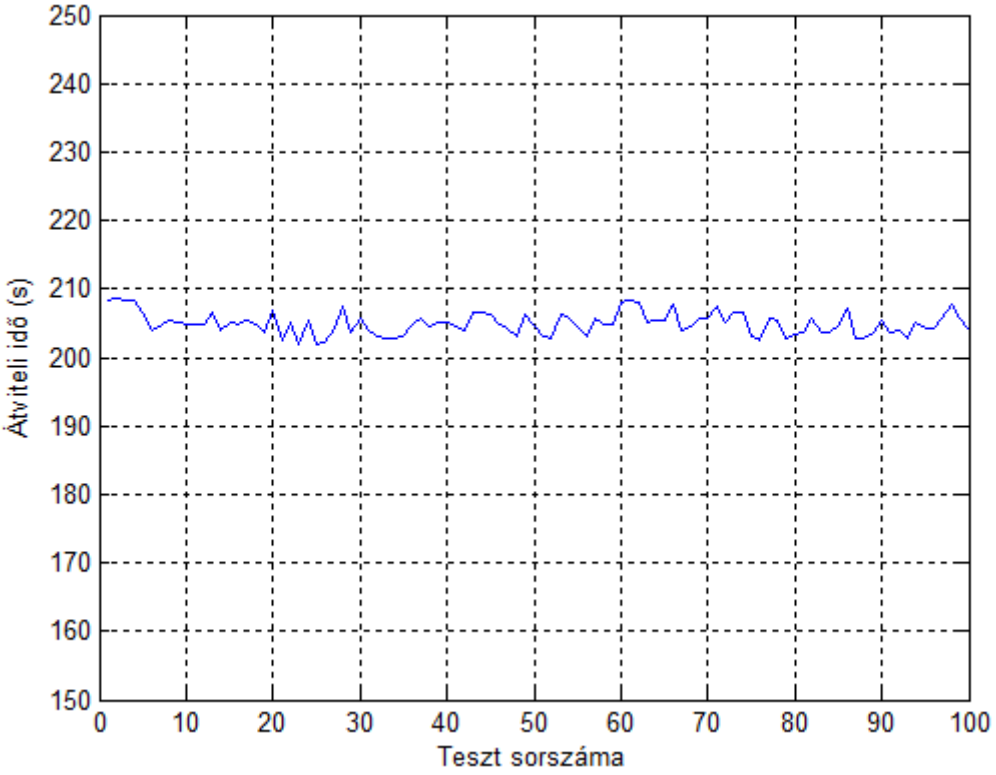
4.3.4. Adatátvitel feltételes forgalomszabályozással

Bemutatom a probléma megoldását kétoldali feltételes forgalomszabályozás segítségével is. Mindez azt jelenti, hogy egyrészt a küldő értesítést ad új adat érkezéséről, másrészt a fogadó visszajelzést tud adni, hogy az adatot elvette, és kéri a következőt. Ezekre a funkciókra is folyamatokat, Service-eket használunk, így bár az adatátvitel biztosítva lesz, még több erőforrásra lesz szükségünk, valamint a fogadó alkalmazásnak is szüksége van az Intent használatára. Ezek a Service-ok egyszerű jelzések lesznek, SENT és ACK megnevezéssel. A SENT az új, érvényes adat létrejöttét jelzi a fogadó alkalmazásnak. Amint a fogadó látja ezt, beolvassa az adatot, és az ACK (acknowledged) jelzéssel jelez, hogy megkapta a küldött információt. Ennek hatására mind a SENT, mind az adatot jelentő process leáll, végül az ACK is megszűnik, és kezdődik az egész elölről. Természetesen a lekérdezés nem folyamatos, hiszen az rettentő processzorigényes lenne, így ha nem talált az egyik alkalmazás neki megfelelő folyamatot a listában, egy kis idővel később fogja azt újra megnézni. A működés folyamatábrája az 5. ábrán látható.

A feltételes forgalomszabályozás hatására biztosítva van az adatátvitel, ezt a tesztesetek is megerősítik. Azonban nem szabad megfeledkeznünk a módszer hátrányairól: egyrészt nagyon sok folyamat indítását és leállítását igényli, amely igen erőforrás-igényessé válhat. Mivel nincs időzítés, a ps-t folyamatosan, periodikusan le kell kérdeznünk, ráadásul ezt itt az adatküldő folyamatoknak is meg kell tenniük, hiszen nem időre, hanem az ACK megjelenésére kell leállniuk, ezt ők is csak a ps-ből tudhatják meg.



5. ábra. Adatátvitel folyamatokkal forgalomszabályozás esetén



6. ábra. 100 szám átviteli idejének tesztelése feltételes forgalomszabályozással

Mindezt konkrét, valós számadatokkal is alátámaszthatjuk: száz szám átvitele itt már mintegy 200 másodpercbe kerül, ahogy ezt a 6. ábrán is láthatjuk, tehát az előző megoldáshoz képest feleannyi lett az adatátvitel sebességünk. Ezt a sebességet áldoztuk fel a megbízhatóságért. Ezzel a megoldással a 16 szám átvitele hozzávetőlegesen fél perc alatt megvalósítható, ami továbbra is megfelelő érték ilyen mennyiségű adat esetén. Mindkét megoldásnak megvan tehát az előnye és hátránya is, így a valós működési környezetnek megfelelőt érdemes alkalmazni.

4.4. A fájlkezelés

Utolsó alternatív megoldásunkban a fájlkezeléssel foglalkozunk. Megismerkedünk a fájlkezelés, a különböző fájlok elérésnek alapjaival. Ezt követően az Android által biztosított módszereket vizsgáljuk, illetve egy, a folyamatokkal való kommunikációhoz hasonló side-channel megoldást ismertetünk.

4.4.1. A fájlkezelés Linuxon

Eddig is láthattuk, hogy az operációs rendszer alapját képező Linux kernel az Android szinte minden szegmensét áthatja. Nincs ez másképp a fájlkezelés területén sem. Az alkalmazások szeparációjának következményeként alapesetben különböző applikációk nem férhetnek hozzá egymás fájljaihoz. A Linux (pontosabban a UNIX-alapú rendszerek) egy jogosultságrendszert használnak ennek kezelésére. Három osztályt különböztetünk meg: ezek a user, a group, és az others. A user a fájl (vagy könyvtár) létrehozója, a hozzá

tartozó jogosultságok csak erre az egy felhasználóra vonatkoznak. A group felhasználók egy csoportja, például egy webszerver esetén több embernek is jogosultságot biztosíthatunk bizonyos fájlok eléréséhez. Az others, ahogy a neve is mutatja, mindenki mást jelent: mindenki, aki nem tartozik egyetlen csoportba sem (és nem a fájl létrehozója), az ehhez az osztályhoz tartozó jogosultságokkal fog rendelkezni.

Mindhárom osztály különböző engedélyekkel rendelkezhet egy-egy fájl eléréséhez, használatához, módosításához: ezek a read, write, execute, tehát olvasás, írás, végrehajtás műveletek. Minden osztály ilyen típusú jogosultságai beállíthatóak mindegy egyes fájl esetén. A read olvasáshoz ad jogot: egy fájl olvasásához, vagy egy könyvtárban található fájlok neveinek olvasásához használható. A write fájlok írását, illetve könyvtárakban fájlok létrehozását, törlését, átnevezését teszi lehetővé. Az execute parancs szükséges futtatható fájlok végrehajtásához, könyvtár esetén pedig a könyvtár tartalmáról kérhetünk információt a segítségével.

Láthatjuk, hogy az applikációk megfelelő szeparációja esetén tulajdonképpen azoknak esélyük sem lehetne ilyen formán információt átvinni: nemhogy egy tetszőleges fájlt tartalmát nem képesek kiolvasni, de még a fájlok létezését is el lehet rejteni mások elől. Ez valóban egy jó biztonsági megoldást feltételez, amely a Linux esetén be is vált, azonban nézzük meg mindezt a gyakorlatban.

4.4.2. Az fájlkezelés alkalmazásainkból

Tekintsük először az Android által biztosított eszközöket fájlok eléréséhez, módosításához. Fájlok megnyitására az openFileInput() metódussal van lehetőségünk: az ennek paraméterül átadott fájlnév megnyitását kísérli meg a rendszer, amennyiben ez sikeres, a fájl tartalmát kiolvashatjuk a metódus által visszaadott adatfolyamból, azonban ha sikertelen volt a megnyitás, FileNotFoundException-t kapunk, akkor is, ha a fájl létezik, de nincs jogosultsága az applikációnak annak megnyitására.

Fájlok létrehozására az openFileOutput() áll rendelkezésünkre: első paramétere az openFileInput-hoz hasonlóan egy fájl neve (amely ha nem létezik, automatikusan létrehozza), a második pedig a létrehozás módjára vonatkozik. Ezek a módok a következők: `MODE_APPEND`, mellyel egy létező fájl végére írhatunk, így a benne lévő adat nem vész el; `MODE_PRIVATE` esetén mások számára nem lesz elérhető a fájl, csak az azt létrehozó alkalmazásnak; `MODE_WORLD_READABLE` és `MODE_WORLD_WRITEABLE` esetén mindenki számára olvasható/írható lesz a fájl. Azonnal észrevehetjük, hogy utóbbi két mód az operációs rendszer biztonsági architektúrájának egyik legfontosabb részét képző applikáció szeparációt jelentős mértékben megsérti. Nincs értelme ugyanis a szeparációnak, ha korlátlanul létrehozhatunk fájlokat, melyeket mindenki tud olvasni, esetleg írni, ezáltal ugyanis bármilyen adatot átadhatnak egymásnak.

Azonban ezt természetesen a gyakorlatban is ki kellett próbálnunk, hátha valamilyen korlátozás vonatkozik ezen eszközök használatára. A tesztelő alkalmazások nagyon egyszerű felépítéssel rendelkeznek: az egyik létrehoz egy fájlt `MODE_WORLD_READABLE` módban, a másik alkalmazásban ezt a fájlt pedig megpróbáljuk beolvasni openFileInput segítségével. A fájl nevét nem kell átadnunk futás közben az olvasó alkalmazásnak, mivel mi vagyunk a fejlesztők: egyszerűen "bedrótozhatjuk" az adott fájl nevét az olvasó alkalmazásba is, hiszen minden eszközön ugyanott fog ez a fájl létrejönni, így a hordozhatóságot nem befolyásolja. Ezt követően a kipróbálás a várakozásoknak megfelelő eredménnyel zajlott le: valóban képes az olvasó applikáció, ami elvileg teljesen szeparálva van a másiktól, kiolvasni a fájl tartalmát. Az egyetlen jelzés a fejlesztőkörnyezet részéről egy megjegyzés volt a fájlmegnyitás sorában, miszerint nem javasolt ennek a megnyitási módnak a hasz-

nálata, azonban ezen kívül semmi más problémába nem ütköztem, gond nélkül futtatható mindkét alkalmazás. Láthatjuk, hogy a fejlesztő kezében az Intent után egy újabb olyan eszköz van, mely a szeparáció, így a biztonsági megoldások teljes megkerülésével adattovábbításra képes két applikáció között. Mindez egy olyan rendszer esetén, melyre bárki fejleszthet alkalmazást, és közzéteheti azt, ilyen könnyű megvalósítás esetén meglehetősen kifogásolható.

A privát megnyitás tesztelését is elvégeztem, hogy megbizonyosodjak arról, hogy valóban jó a működése és nem lehet más alkalmazásból az így létrehozott fájlokat megnyitni. A vizsgálat elvégzéséhez minél több rendelkezésre álló eszközt felhasználtam, hogy minél nagyobb bizonyossággal lehessen azt mondani a módszerről, hogy valóban biztonságos, és ez a biztonsági megoldás megfelelő. Ehhez nézzük meg, milyen eszközöket kínál fel maga a Java nyelv számunkra. Itt megjegyezzük, hogy a fájlkezelési módszerek a Java Development Kit (JDK) 7-es, új verziójában kissé megváltoztak, azonban az Android SDK még csak a 6-os JDK-t, támogatja, így a dolgozatban az ebben megtalálható módszereket mutatjuk be. Ahogy a dolgozat korábbi részén is láthattuk, két pont közötti kommunikáció Java-ban adatfolyamokkal, stream-ekkel valósul meg. Fájlok esetén is írásra és olvasásra ezek az eszközök állnak rendelkezésünkre: FileInputStream segítségével beolvashatunk egy fájlból, míg FileOutputStream segítségével kiírhatunk egy fájlba adatokat. Ezen kívül a Java API része még a File osztály, mellyel egy File-t objektumként kezelünk, a fájl nevével konstruálhatjuk az objektumot, ezt követően különböző információkat kérdezhetünk le: mi a fájl teljes elérési útja, mekkora a mérete, létezik-e ez a fájl stb.

A kipróbálás során tehát immár private módban hozzuk létre a fájlt. Az openFileInputtal történő megnyitás esetén nem ér minket meglepetés: FileNotFoundException-t kapunk, tehát fájl létezéséről sem tud a program, így ez a mód első látásra valóban jól működik. Kipróbáltam a FileInputStream-mel történő megnyitást is, ugyanaz lett az eredmény, FileNotFoundException miatt sikertelen lett a megnyitás. Ekkor azt gondolhatnánk, hogy ezzel a megoldással így nincs probléma, amennyiben privát módon történik a fájl megnyitása, a tartalmát biztosan nem tudjuk kiolvasni. Mindez így is van, azonban egy utolsó próbálkozást még tettem. A File osztály segítségével létrehoztam egy objektumot a privát fájl nevével. Leellenőriztem, hogy létezik-e a fájl, tehát a File osztály exists() metódusa milyen értékkel tér vissza. Meglepő eredmény született: a privát fájl létezésének tényét ezzel a módszerrel meg tudjuk állapítani, ugyanis true, igaz értéket kaptam, míg nyilvánvalóan, ha ez a fájl nem létezik, hamis lesz a függvény visszatérési értéke. Mindez új lehetőségeket tár fel az alternatív megoldások körében: amennyiben ezt a tényt fel tudjuk használni valamilyen formán információ küldésére, akkor a szeparáció fenntartása mellett (tehát `WORLD_READABLE` mód használata nélkül, privát fájlokkal) vagyunk képesek adatátvitelre, mely ismételten a biztonsági architektúra gyengeségét jelzi.

4.4.3. Kommunikáció megvalósítása fájlok használatával

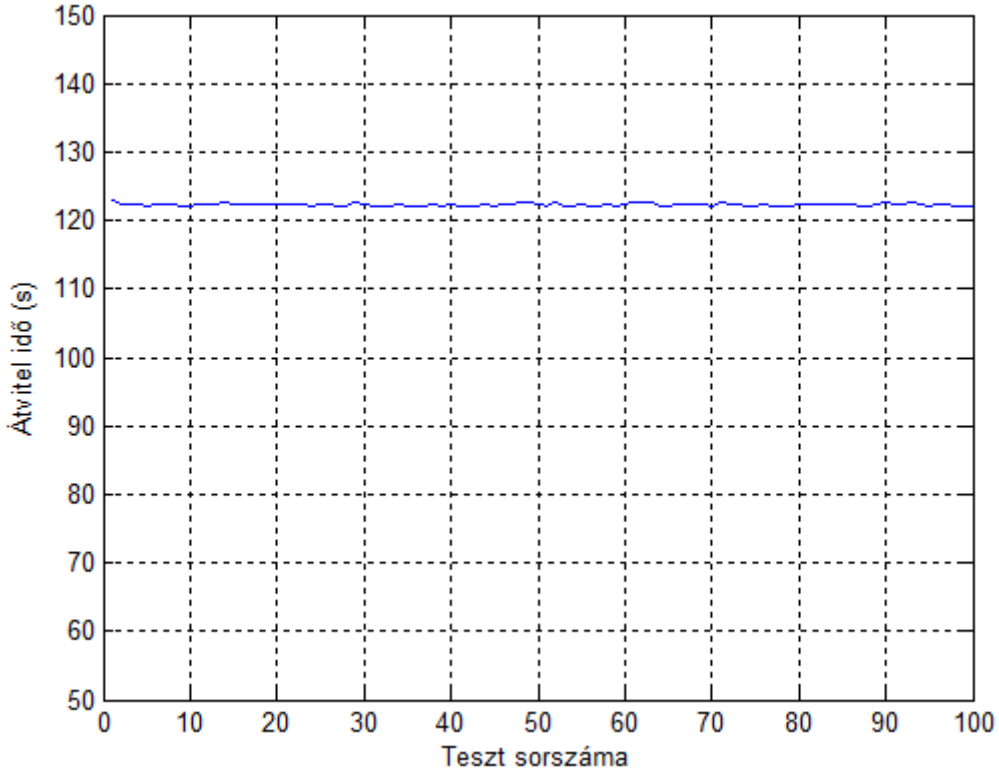
Látjuk tehát, hogy bármilyen jogosultsággal is rendelkezzünk egy fájl használatához, annak létezését (ha tudjuk a nevét), meg tudjuk állapítani. Ezt a tényt fogjuk felhasználni kommunikáció megvalósítására. A módszer nagyon hasonló lesz a processekkel történő kommunikációhoz, azonban több ponton is különbözni fog a megvalósítás. Az alapötlet itt is az, hogy bináris adat küldését végezzük el, a két küldendő bitet két fájl reprezentálja. A küldő fél létrehozza a megfelelő névvel az adathoz tartozó fájlt, ezt érzékeli a fogadó, hiszen a fájl létezését detektálni tudja. Ezután a fájlnév alapján tudja, melyik bit érkezett, ezt el tudja menteni, és kezdődik az egész folyamat elölről. Itt is meg kell vizsgálnunk a forgalomszabályozás és az adatátvitel sebességének kérdését.

A forgalomszabályozás során - hasonlóan a folyamatok második megoldásánál - kétoldali feltételes forgalomszabályozást alkalmazunk. Itt ugyanis jóval kevesebb hátránnyal kell szembe néznünk: egy-egy fájl létrehozása és törlése a processzor és az operációs rendszer számára jóval kevesebb munkával jár, mint egy Service elindítása, majd annak leállítása. Így az adatátvitel nagy megbízhatósággal fog rendelkezni, amelyért mindenképp kevesebb értékes erőforrást kell feláldoznunk.

Az adatátviteli sebesség kérdése is lényeges: nyilvánvalóan minél nagyobb sebességre törekszünk, hogy minél gyorsabban véget érjen egy-egy küldési ciklus, hiszen így nem kockáztatjuk azt, hogy az átvitel során az egyik fél leállításra kerül az operációs rendszer által, amelynek következtében megszakad az adatok továbbítása. A folyamatok esetén a szimbólumrendszer bővítése sokat segített, azonban ott egy teljes lista rendelkezésre állt a futó folyamatokról, és ebből a listából csak ki kellett keresnünk a megfelelő nevű processt. Ilyen lista esetén (a processekkel ellentétben) akár nagy mennyiségű adatot is nagyon gyorsan át tudnánk vinni ebben az esetben, hiszen a létrehozott fájl nevében elhelyezhetjük ezeket az információkat, amelyeket a fogadó oldalon visszanyerhetünk. Azonban itt ilyenre nincs lehetőségünk. Bár a File osztály képes könyvtárakat is reprezentálni, ennélfogva rendelkezik olyan metódussal, mely az adott könyvtár tartalmát kilistázza, esetünkben nincs lehetőségünk erre, ugyanis a létrehozott fájlokat tartalmazó könyvtáron erre nincs jogosultsága a programunknak. Mindez azt jelenti, hogy amennyiben decimális számok átvitelét szeretnénk végrehajtani, minden ilyen számjegyre végig kéne néznünk, hogy létezik-e az adott fájl, tehát megkonstruálni a megfelelő objektumot, és ellenőrizni a fájl létezését. Mindez meglehetősen körülményes, így bináris átvitellel oldottam meg az adatok továbbítását, így bár egy decimális szám átvitele négy bitbe kerül, a fájlok létrehozásának és törlésének nagy sebessége miatt mégis egy megfelelően gyors megoldást kapunk.

Az implementáció tehát a fenti séma szerint valósul meg. A forgalomszabályozás szintén SENT és ACK jelekkel valósul meg, ahogy azt a processeknél is láthattuk, de itt természetesen fájlok formájában jönnek létre, ahogy a küldendő adat is fájl formájában jelenik meg. A megvalósítás tehát a következő: a küldő alkalmazás a megfelelő adatot (legyenek ezek decimális számok, ha az eredeti példánál maradva földrajzi koordináták átvitelét szeretnénk megvalósítani) felbontja 0-1 sorozatokra, esetünkben BCD kódra (lehetne egyszerűen kettes számrendszerbe átváltani, de mivel a tesztek alapegysége korábban a decimális szám volt, így itt is ezt alkalmaztam). Ezt a sorozatot bitenkénti küldi el a fogadónak, a következőképpen: létrehozza az adott bithez tartozó fájlt, illetve a SENT jelet is kiadja (szintén a fájl létrehozásával). A SENT fájl létezését képes detektálni a fogadó alkalmazás, ennek hatására a 0 vagy 1 bitet beolvassa, annak függvényében, hogy melyik fájlt találta meg. Beolvasás után az ACK fájl létrehozásával visszaigazolja a bit vételét. Ezt érzékeli a küldő oldal, elveszi a SENT jelet és az adatot, a SENT megszűnésére pedig a fogadó törli ki az ACK fájlját, és mindez kezdődik elölről. Jól látható, hogy teljesen analóg a megoldás a folyamatoknál látottakkal, mindössze itt fájlok létrehozása és törlése van, folyamatok indítása és leállítása helyett.

A megbízhatóságot tehát biztosítják a forgalomszabályozó jelzések. Azonban tekintsük az adatátviteli sebességet: azt várhatnánk, hogy a processeknél látott körülbelül 2 másodperc/szám-nál lassabb sebességet tudunk elérni, hiszen itt 4 bit küldését kell elvégeznünk egyetlen decimális szám átviteléhez. Azonban a fentebb is említett okok miatt nem ez a helyzet: a folyamatok elindítása és leállítása jóval lassabb, mint fájlok létrehozás és törlése. Mindezt a tesztadatok is alátámasztják, ahogy azt a 7. ábrán is láthatjuk: száz szám átvitelét (ez 400 bitet jelent) mintegy 2 perc alatt el tudunk végezni, tehát egy



7. ábra. 100 szám átviteli idejének tesztelése feltételes forgalomszabályozással

számot alig több mint 1 másodperc alatt tudunk átvinni. Ez valóban egy számunkra megfelelő eredmény, hiszen itt az adatátvitel megbízható, és közel olyan sebességet értünk el, mint amit a folyamatokkal való kommunikáció forgalomszabályozás nélküli megoldásánál.

5. Megoldási javaslatok a hibákra

Az eddigi fejezetekben megismerkedtünk több Inter-Process Communication lehetőséggel. Bizonyos megoldások az Android fejlesztőeszközeinek segítségével valósulhatott meg, illetve nem szokványos, alternatív kommunikációs formákat is bemutattunk. Ezek a megoldások a felhasználó személyes adatainak biztonságát sértik, fontos tehát, hogy minél előbb kijavításra kerüljenek. Nyílt rendszerről lévén szó, rengeteg fejlesztővel rendelkezik az Android, akik folyamatosan újabb és újabb szoftvereket adnak ki. Egy ilyen rendszer esetén a fejlesztőket érdemes minél nagyobb kontroll alatt tartani, minél inkább korlátozni a lehetőségeiket, hogy ne élhessenek vissza az alkalmazásikkal. Természetesen az operációs rendszer fejlesztője, a Google számára is rendkívül fontos a privát adatok biztonsága. Az általuk kifejlesztett alkalmazásszűrővel, a Google Bouncerrel is megismerkedünk ebben a részben, valamint az IPC megoldások által kihasznált hibák javítására is javaslatot teszünk.

5.1. A Google Bouncer

A Google is tisztában van a nyílt rendszerek előnyeivel és hátrányaival, így jól tudják, hogy előbb-utóbb valaki nem megfelelően fogja használni az általuk biztosított eszközöket. Ennek kivédésére fejlesztették ki az úgynevezett Google Bouncert [5]. A rendszer a Google Play-re feltöltött alkalmazások vizsgálatát végzi el. Egyrészt a már feltöltött applikációkban, másrészt az újonnan feltöltöttekben keres vírusra, malware-re, trójaira, illetve egyéb károkozókra utaló jeleket. Amennyiben gyanús viselkedést észlel, további alkalmazásokhoz hasonlítja a kétes applikációt, és felderíti a potenciális veszélyforrásokat. Mindez egy felhős rendszer segítségével valósul meg, tehát különböző eszközök hálózatba kapcsolva használhatják egymás erőforrásait. A vizsgált alkalmazások ezen a rendszeren futnak, itt figyelik a működésüket, egymással való kooperációjukat.

Természetesen a pontos vizsgálati módszerekről nincs információ, hiszen akkor könnyű lenne a Bouncer kijátszása. A rendszer nagy előnye, hogy valós szimulációs körülményeket teremt, valamint sok alkalmazás egyszerre futtathatóvá válik, így együttműködésük könnyen vizsgálható. A Google Bouncer továbbá nem csak az alkalmazásokat, hanem a fejlesztőket is figyeli, így egy-egy gyanús alkalmazást író fejlesztő kitiltását automatikusan megoldhatja. A rendszer hatására hivatalos információk szerint 2011-ben mintegy 40%-kal csökkent a károkozó alkalmazások száma a Google Play-en, mindez a vírusirtó és biztonsági applikációk megjelenésének is köszönhető volt. Egyes információk szerint a Bouncer kijátszásával már többször sikerrel jártak [6], azonban mindenképp fontos megemlíteni a rendszer meglétét, hiszen bizonyos, a dolgozatban vizsgált megoldások akár már rögtön a feltöltés után elbukhatnak, ha a Bouncer megfelelően van kialakítva, azonban erre különösen az alternatív megoldások esetén kicsi az esély. Az általam felvázolt megoldások nem kerültek megvizsgálásra, de ez a jövőben mindenképp meg fog történni.

5.2. Konkrét megoldási lehetőségek a bemutatott problémákra

A rendszer biztonságának fenntartása érdekében a felmerülő problémákat minél előbb érdemes orvosolni, hogy azok kihasználást megakadályozzuk. Ebben a részben a korábban bemutatott IPC lehetőségek meggátolására adunk javaslatokat, ötleteket a teljesség igénye nélkül. Az első, legfontosabb problémát az Intentek jelentik: széleskörűen elterjedt eszközökről van szó, amelyekkel nagyon egyszerűen valósíthatjuk meg az adatátvitelt különböző applikációk között. Ebben az esetben a megoldást az jelenthetné, ha telepítéskor

megvizsgálná az operációs rendszer legalább az explicit Intenteket: mindezt a telepítendő applikációnál és a már telepítetteknél is. Amennyiben úgy találná, hogy explicit Intenthívás történhet a telefonon az új alkalmazás közreműködésével a telepítés után, akkor a hívott alkalmazások jogosultságaival összevetheti a telepítendő alkalmazás által kért erőforrásokat. Amennyiben különböző jogosultsággal rendelkező alkalmazások hívják egymást, erről értesítheti a felhasználót az operációs rendszer, vagy akár egyből a telepíteni kívánt alkalmazásnak is rendelkeznie kell a kérdéses engedélyekkel, így a felhasználót nem vezethetik félre ezek az applikációk.

Az implicit Intentek esetén precízen nem lehet mindig beazonosítani a hívott alkalmazást, azonban, ahogy azt korábban is megemlítettük, a fejlesztő speciális feltételekkel is létrehozhat implicit Intentet, amelyeknél a célalkalmazás már egyértelműen azonosítható, így a fenti eljárás itt is elvégezhető.

A következő problémát a Socket jelentette. Itt, bár mindkét alkalmazás kért jogosultságot a hálózat használatára, a korrekt ellenőrzés hiánya továbbra is fennáll. A felhasználót offline használat esetén félrevezetheti a jogosultságok megléte, így az applikációk nem állnak megfelelő kontroll alatt, melyet a biztonsági rendszer előnyének tekintenek. A problémára talán a legegyszerűbb megoldást a loopback address ilyen módú használatának letiltása jelentheti, melyet megvalósítani sem lenne nehéz: Socket osztályt egyszerűen nem lehet erre a címre konstruálni, ezáltal ellehetetlenítve ennek a hibának a kihasználását.

Ezt követően a signalokat vizsgáltuk, viszont itt sajnos nem jártunk sikerrel, a signalok küldése más applikáció számára le van tiltva. Azonban az ebből a vizsgálatból létrejövő, folyamatokkal történő kommunikáció teljesen működőképes, és az Intentek fentebb javasolt fokozott ellenőrzésével is használható maradna a megoldás, főleg, hogy Intentek hívása más alkalmazás irányába nem is történik, csak saját applikáción belül. Mivel az egész működés a ps parancson alapul, így annak letiltása teljesen megoldaná a problémát. Viszont ha bizonyos oknál fogva szükség van erre a parancsra, egy másik lehetőség, hogy a folyamatok nevének beállítását nem lehetne megtenni a Manifest fájlban: mivel az adatátvitel a folyamatok neveinek segítségével valósul meg, így ha ezek beállítására nem lenne lehetőség, a teljes applikáció egy folyamatként futna, így a kívánt működést nem lehetne megvalósítani. Valószínűleg egyik módszer megvalósítása sem bonyolult, így ezt a hibát könnyedén ki lehet javítani.

Az utolsó részben a fájlkezeléssel foglalkoztunk. Ezen a területen több hiba kihasználása is lehetséges: elsőként a fájlok létrehozásánál használt `MODE_WORLD_READABLE` flag működését vizsgáltuk, és láthattuk, hogy az applikáció szeparációval teljesen ellentmond ez az eszköz az Android SDK-ban. Valószínűleg a legjobb megoldás ennek a módnak (és a `MODE_WORLD_WRITEABLE`-nek is természetesen) a megszüntetése, hiszen így lehetősége sem lenne a fejlesztőnek ilyen egyszerűen adatokat átvinnie két alkalmazás között. A második megoldásunk egy igen különleges hibát használ ki, miszerint egy védett, privát módon létrehozott fájl létezését bármely applikációból meg tudjuk állapítani. Láttuk, hogy bár a különböző, Java által biztosított fájlmegnyitási eszközök, valamint az Android openFileInput()-ja is kudarcot vallott, a File osztállyal mégis lehetőségünk van erre. A megoldást ennek az osztálynak a felülvizsgálata jelentheti, a kérdéses metódust kell valamilyen módon korlátozni, vagy blokkolni.

6. Kapcsolódó irodalom

Ebben a fejezetben bemutatom a témával kapcsolatos aktuális irodalmat, publikációkat, cikkeket. Elsőként tekintsük a dolgozatban is felhasznált David Wagner és társai által készített Analyzing Inter-Application Communication in Android című cikket [8]. Ahogy a cím is mutatja, szintén applikációk közötti kommunikációval foglalkoznak. Az Androidos alkalmazások együttműködését vizsgálják, és felderítik a biztonsági kockázatot jelentő problémákat az alkalmazások különböző komponenseiben. Míg dolgozatom a kommunikációs lehetőségek szélesebb területét próbálja feltérképezni, addig a cikk főképp az Intentek területére, itt is az implicit Intentekre koncentrál. Bemutatásra kerül az applikációk alapvető felépítése, az Intenthívási lehetőségek, majd a dolgozatban is megemlített problémák, pl. a Broadcast eltulajdonítása, vagy az Activity Hijacking részletes leírása. Létrehoztak egy eszközt, a ComDroid-ot, mellyel az alkalmazások által kezdeményezett kommunikáció sérülékenységét vizsgálják. Ennek segítségével a fejlesztők felülvizsgálhatják alkalmazásaikat, és a potenciális veszélyforrásokat kijavíthatják még a publikálás előtt. Húsz alkalmazás vizsgálatát a szerzők is elvégezték, 34 veszélyforrást találtak, 12 applikációban legalább egy hiba volt.

Szintén David Wagner és társai készítették az Android Permissions Demystified című cikket [9]. Ebben az írásban a jogosultságok rendszeréről kapunk egy részletes leírást. A cikk készítése során különböző alkalmazások vizsgálatát végezték el, annak kiderítésére, hogy a fejlesztők milyen szabályokat követnek az egyes jogosultságok igénylésénél. Létrehoztak egy programot, a Stowaway-t, amely automatikusan feltérképezi az egyes alkalmazások rendszerhívásai alapján azokat a jogosultságokat, amelyekre nem is lenne szüksége az applikációnak. 940 alkalmazáson végeztek teszteket, melyeknek körülbelül egyharmada rendelkezett ilyen jogosultságokkal.

Ugyancsak a Berkeley University munkatársai által kiadott cikk a Reducing Attack Surfaces for Intra-Application Communication in Android [11]. Az írás arra fókuszál, hogy a különböző applikációkban számos sebezhetőségi pontot jelent az, hogy a fejlesztők Inter-Process Communicationt használnak akkor is, amikor Intra-Process Communication is elegendő lenne, tehát az adott folyamatban maradva, belső eszközökkel is megvalósítható lenne az átvitel. Szintén egy automatizált eszköz segítségével próbálják az olyan alkalmazások közötti üzeneteket detektálni, amelyeknek alkalmazáson belüli üzeneteknek kellene lenniük. Képesek lettek a korábbi munkájuk során talált Intra-Application sérülékenységek 100%-át kijavítani, és az Androidos alkalmazások több, mint 90%-a kompatibilis ezekkel a javításokkal.

Szintén David Wagner és társai írták az AdDroid: Privilege Separation for Applications and Advertisers in Android [13] című cikket, melyben ismét a jogosultságokról van szó, azonban egy más kontextusban: a hirdetési tevékenységet is folytató alkalmazások gyakran használnak külön erre a célra kifejlesztett függvénykönyvtárakat. Ezek közös jogosultságokkal rendelkeznek, így a főprogram által elkért, privát adatokhoz is hozzáférést biztosító jogosultsággal a hirdető könyvtár is rendelkezik. Eredményeik alapján az applikációk mintegy fele rendelkezik ilyen könyvtárral, és sok jogosultságot csak a hirdetési tevékenység miatt kérnek el, például helymeghatározást csak a hirdetések érdekében végeznek bizonyos alkalmazások. A problémára a szerzők által fejlesztett AdDroid adhat megoldást: ez egy olyan keretrendszer, melyben a hirdetés a főprogramok működésétől különválik, így felhasználói biztonságot veszélyeztető jogosultságok kérésére nincs szükség.

Sven Bugiel és társai a Practical and Lightweight Domain Isolation on Android [7] című

cikkben az applikációk közötti kommunikáció meggátolására fejlesztették ki a TrustDroidot. Segítségével a rendszert három rétegre bontják szét: az egyik ilyen réteg a rendszermag, a fájlrendszer és különböző IPC lehetőségek meggátolására; a második egy középső réteg az alkalmazások kommunikációjának megakadályozására; míg a harmadik a hálózati réteg a hálózati kommunikáció biztosítására. Mindez egy erőforrástakarékos megoldás, mely a felhasználói biztonság fokozását segíti elő.

David Barrera és társai a user ID rendszerrel foglalkoznak az Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android cikkben [12]. Bemutatják a signing hátrányait, illetve azt, hogy a fejlesztőket nagyban korlátozza a UID rendszer, hiszen akár az operációs rendszer is elvégezhetné az applikációk közötti kommunikáció ellenőrzését, biztosítását. Ezt követően a cikkben bemutatnak egy olyan biztonsági hibát, mellyel az egyes alkalmazások több jogosultságot is kaphatnak, mint amennyit a Manifestben elkérnek.

A Berkeley University munkatársai a jogosultságok felhasználókkal való kapcsolatát mutatják be a Android Permissions: User Attention, Comprehension, and Behavior című cikkben [10]. Azt vizsgálják, hogy a felhasználók tisztában vannak-e a különböző alkalmazások által kért jogosultságok következményeivel, megfelelően figyelmezteti-e a felhasználókat a rendszer jogosultság bemutatásával a telepítés során. Két felmérést is végeztek, összesen 333 ember részvételével, amiből az derült ki, hogy a felhasználóknak csak kis része figyel oda ezekre telepítés során, és mindössze a megkérdezettek 3%-a van tisztában az egyes jogosultságok pontos jelentésével. Erre a problémára próbálnak megoldást, javaslatokat nyújtani a cikk szerzői.

Timothy Vidas és társai az All Your Droid Are Belong To Us: A Survey of Current Android Attacks [14] cikkben az Android általános veszélyforrásairól írnak, mivel ez a közelmúltban egy jelentős támadási felületté vált. Bemutatják az Android biztonsági modelljét, és ennek a modellnek bizonyos hiányosságait. Ezt követően olyan módszerek kerülnek bemutatásra, melyek a valóságban is tesztelésre kerültek, és hozzáférést lehet nyerni velük Androidos készülékekhez. A felfedezett hibákhoz, ahol lehetséges, megoldási lehetőségeket is javasolnak a cikk szerzői.

7. Konklúzió

Dolgozatomban az Android operációs rendszeren applikációk közötti kommunikációk lehetőségét vizsgáltam meg. Mindez azért fontos, mert amennyiben képesek erre a kommunikációra az alkalmazások, szenzitív felhasználói adatok kiszivárogtatását végezhetik el. Ráadásul ezt a felhasználó félrevezetésével teszik, hiszen a jogosultságrendszer biztonságérzetet ad ezekben az esetekben, a felhasználó úgy érzi, irányítása alatt tartja készülékét, miközben az egyes alkalmazások adatot cserélnek, ezáltal felhasználóhatóak lesznek károkozás céljára. Napról napra egyre több Androidos készüléket adnak el, így minél hamarabb fel kell derítenünk ezeket a lehetőségeket, hogy kijavításra kerülhessenek. A dolgozatomban a rendszer rövid bemutatását követően egyrészt az Android által biztosított eszközökkel megvalósítható IPC lehetőségeket, másrészt alternatív, nem triviális megoldások vizsgálatát is elvégeztem.

Bizonyos ötletek vizsgálata végén kiderült, hogy nem, vagy nem tökéletesen alkalmas az elképzeléseknek megfelelő működésre a megoldás. Ilyen terület volt például a sharedUserId. Ennek az eszköznek a biztonsági megoldásai megfelelőek, így az általunk elérendő cél elérésére nem volt alkalmas. Fontos azonban a lehetőségek minél szélesebb skálájának bejárása, hiszen csak így adhatunk átfogó képet a rendszer ezen aspektusáról, ezért elengedhetetlen volt ennek az eszköznek a vizsgálata. A Linuxos alapoknak köszönhetően az operációs rendszer gazdag eszközkészlettel rendelkezik a különböző folyamatok közötti kommunikáció területén. Ide tartozik a signal is, ami bár hasonló eredménnyel járt, mint a sharedUserId, mégse olyan triviális a vizsgálata, mint az előbbi eszköznek. A signal különböző applikációk között a jelek szerint le van tiltva, így bár remek lehetőség lett volna kommunikáció megvalósítására, ezen a területen nem értünk el sikereket. Láthattuk viszont, hogy később egy új lehetőséghez vezetett a signalok küldésének feltérképezése. Szintén tradicionális megoldás a UNIX-rendszereken a socket. Ez már esetünkben is működőképes volt, segítségével az alkalmazások a loopback addressen keresztül sikeresen kommunikálnak egymással. Azonban ez a megoldás sem tökéletes: a jogosultságrendszer ebben az esetben jól működik, így minden, kommunikációban résztvevő alkalmazásnak a hálózathasználathoz engedélyt kell adni. Ennek ellenére viszont sérti az applikációk szeparációját, amely a biztonsági rendszer egyik alappillére, hiszen internetkapcsolat nélkül is tudnak kommunikálni az alkalmazások, amikor erre alapvetően nem lehetne lehetőségük.

Három ötlet esetén viszont egyértelműen sikerrel jártunk a kommunikáció megvalósításában. A dolgozat fontos része az Intentek tanulmányozása. Láthattuk, hogy az Intent mindkét típusa veszélyt jelent a felhasználói adatok biztonságára. Az explicit Intentekkel a fejlesztő nagyon könnyedén képes adatok átvitelére két alkalmazás között, mindössze a célalkalmazás egyedi azonosítóját kell ismernie. Az implicit Intentek segítségével pedig nem csak adatátvitelre, hanem adatok eltulajdonítására, illetve a felhasználó megtévesztésére is lehetőség van. Az Intentek az alkalmazások közötti kommunikáció legegyszerűbb megvalósítói, minden fejlesztő számára rendelkezésre állnak és könnyű a használatuk. Mivel a kommunikáló alkalmazások bármilyen jogosultságokkal rendelkezhetnek, ez nincs ellenőrizve, az Intenttel a szenzitív adatok nem helyénvaló felhasználását bármely fejlesztő megvalósíthatja, mindez egy ekkora felhasználóbázissal rendelkező rendszerben nem megengedhető.

Szintén sikerrel jártunk a folyamatokkal történő kommunikációval. Ez már egyértelműen egy nem triviális, side-channel megoldás, így a gyakorlatban való előfordulási valószínűsége kicsi, azonban bebizonyítottuk, hogy így megvalósítható a kommunikáció, tehát egy újabb hibát találtunk a biztonsági rendszerben. A megoldás szintén a Linuxnak

köszönhetően jött létre, hiszen az alapja a ps parancs, melyen keresztül a folyamatok elindításával és leállításával valósul meg a kommunikáció. Ebben az esetben szükséges volt két, az adatátvitelt meghatározó tényező vizsgálata: ezek a megbízhatóság és a sebesség. Két módszer is bemutatásra került: az elsőnél egy ad-hoc megoldással, időzítések helyes megválasztásával értük el a megfelelő megbízhatósági szintet, és a gyakorlati működésből származó tesztadatok alapján az adatátviteli sebesség is megfelelő. A másik módszernél a megbízhatóság lett az elsődleges cél, mely egy kétoldali feltételes forgalomszabályozás segítségével lett megoldva. A megbízhatóság így adott, azonban az adatátviteli sebességben fizettük meg ezt az árat, hiszen körülbelül felére csökkent ez a paraméter. A dolgozat szempontjából azonban maga a kommunikáció létrejötte a lényeges, az implementációs kérdésekben a tervezett kommunikáció követelményeinek megfelelően kell eljárni.

Végül a fájlrendszer, a fájlkezelés bemutatására is sor került, hiszen minden alkalmazás hozhat létre fájlokat, és amennyiben sikerülne ezen fájlok elérése más applikációból, akkor könnyedén kommunikálhatnak egymással. Ebben a fejezetben több sikeres megoldást is elkönyvelhetünk. Először a fejlesztőkörnyezet által biztosított fájllétrehozási módokat vizsgáltuk. Meglepő eredmény, hogy az alkalmazások elválasztását egyetlen parancs segítségével felrúghatjuk, és olyan fájlokat hozhatunk létre, amelyek mindenki más által olvashatók vagy írhatók. Természetesen ekkor folytatni kellett a fájlok tanulmányozását, hiszen ennek a hibának a javítása valószínűleg nagyon egyszerű, más lehetőség után kellett néznünk. A Java API fájlkezelő osztályainak vizsgálata során fedeztem fel azt a tényt, hogy egy fájl létezését bármely applikáció megállapíthatja, még akkor is, ha kifejezetten privát módon lett létrehozva a fájl, tehát a szeparáció (elvileg) biztosítva van. Mindez adta az ötletet, hogy a fájlok létezése vagy hiánya adatot reprezentálhat, és így két alkalmazás kommunikálhat. Így született meg a folyamatoknál is látott feltételes forgalomszabályozással megvalósított applikációk közötti kommunikáció a fájlok segítségével. Egy újabb megoldás rendelkezésünkre áll tehát a biztonsági rendszer - és a felhasználó - kijátszására.

Nagy valószínűséggel van még lehetőség alkalmazások kommunikációjára más módon is, dolgozatom mindössze arra szeretne rávilágítani, hogy a rendszer nem eléggé átgondolt, így a felhasználók adatainak biztonsága veszélyben van. További lehetőség lehet a vizsgálódásra újabb kommunikációs csatornák felkutatása, illetve az itt bemutatott módszerek továbbfejlesztése. Különösen a side-channel megoldások sebességének növelése lehet egy ilyen terület, bár természetesen ezek a felhasznált fejlesztői eszközök sebességétől is függnek.

Több módszer segítségével is sikerült demonstrálnunk tehát, hogy az Android biztonsági architektúrájának alapvető elemei, az applikáció szeparáció és a jogosultságok könnyedén kijátszhatóak bárki által. Fontos olyan politikát követni egy mindenki számára rendelkezésre álló fejlesztőkörnyezetnél, hogy a fejlesztők minél kevesebb esélyt kapjanak ilyen rosszindulatú tevékenységek létrehozására. Ezért kiemelten fontos a felfedezett hibák kijavítása, illetve a rendszer folyamatos felülvizsgálása, a fejlesztőknek biztosított lehetőségek ellenőrzése, hogy a felhasználók, és az ő adataik valóban biztonságban lehessenek.

Hivatkozások

[1] Android Developer Page. http://developer.android.com, Letöltve: 2012.október 1.

[2] Android (operating system) - Wikipedia. http://en.wikipedia.org/wiki/Android_(operating_system), Letöltve: 2012.október 1.

[3] Android Source - Security. http://source.android.com/tech/security, Letöltve: 2012.október 1.

[4] Official Android Blog. http://officialandroid.blogspot.com, Letöltve: 2012.október 1.

[5] Official Google Mobile Blog - Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, Letöltve: 2012.október 1.

[6] Researchers beat Google's Bouncer. http://www.net-security.org/secworld.php?id=13333, Letöltve: 2012.október 1.

[7] Sven Bugiely, Lucas Daviy, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghiy, and Bhargava Shastry. Practical and Lightweight Domain Isolation on Android. Technical report, Technische Universität Darmstadt Darmstadt, Germany; Fraunhofer SIT Darmstadt, Germany, 2011.

[8] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. Technical report, University of California, Berkeley Berkeley, CA, USA, 2011.

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. Technical report, University of California, Berkeley Berkeley, CA, USA, 2011.

[10] Adrienne Porter Felt, Elizabeth Hay, Serge Egelman, Ariel Haneyy, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. Technical report, Computer Science Department, School of Information University of California, Berkeley, 2012.

[11] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing Attack Surfaces for Intra-Application Communication in Android. Technical report, University of California, Berkeley Berkeley, CA, USA, 2012.

[12] David Barrera Jeremy Clark Daniel McCarney. Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android. Technical report, School of Computer Science Carleton University, 2012.

[13] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege Separation for Applications and Advertisers in Android. Technical report, University of California, Berkeley Berkeley, CA, USA, 2012.

[14] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All Your Droid Are Belong To Us: A Survey of Current Android Attacks. Technical report, Carnegie Mellon University, 2011.