ELEKTROTEHNIŠKI VESTNIK 93(1-2): 69–78, 2026
IZVIRNI STROKOVNI ČLANEK

Tehnike obhoda zaznave skrbniškega dostopa v aplikacijah Android

Tim Thuma, Tilen Medved, Matevž Pesek

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Večna pot 113, 1000 Ljubljana, Slovenija

Povzetek. Skrbniški način uporabe operacijskega sistema Android lahko prinaša varnostno tveganje za aplikacije, saj omogoča izvajanje zlonamerne programske opreme s povečanimi pravicami, zaradi česar proizvajalci v svoje aplikacije vgrajujejo mehanizme za zaznavo in preprečevanje izvajanja na napravah s skrbniškim dostopom. V tem članku predstavimo štiri načine za obhod tovrstnih zaščit, vključno z dinamično in statično analizo ter dvema konfiguracijama modulov Magisk. Najprej demonstriramo uporabo teh pristopov na testni aplikaciji, nato pa na naboru 23 slovenskih aplikacij analiziramo pogostost zaznave skrbniškega dostopa, vrste implementiranih zaščit ter učinkovitost predstavljenih metod za obhod zaščite. Zaznava skrbniškega dostopa je prisotna v 10 analiziranih aplikacijah, kjer je najpogosteje vključena v bančnih aplikacijah. V petih primerih je obhod zaščite uspešen z vsemi opisanimi metodami, v enem pa le z ročno analizo. V aplikacijah, ki uporabljajo šifriranje kode, je obhod zaščite neuspešen, pri štirih od petih preostalih pa trivialen.

Ključne besede: Android, skrbniški dostop, obhod zaščite mobilnih aplikacij, dinamična analiza, statična analiza, Magisk, Zygisk, Frida, Play Integrity API

Methods to bypass the root detection mechanism in Android applications

The root access on the Android operating system can pose a security risk to applications as it allows the execution of malicious software with elevated privileges. For this reason, manufacturers incorporate mechanisms into their applications to detect and prevent execution on devices with root privileges. The paper presents four methods to bypass such protections, including dynamic and static analysis and two configurations of the Magisk modules. First, it demonstrates their use on a test application, and then it analyzes a set of 23 Slovenian applications to determine the frequency of the root detection, the types of the implemented protection methods, and the effectiveness of the presented protection bypassing methods. The root detection is present in ten of the analyzed applications, where it is most often included in mobile banking applications. In five cases, the protection bypass is successful with each of the described methods, and in one case the protection is bypassed only with a manual analysis. In the applications that encrypt parts of their code, bypassing the protection is unsuccessful, while in four of the five remaining cases, it is trivial to prevent root detection.

Keywords: Android, administrative access, bypassing mobile application protections, dynamic analysis, static analysis, Magisk, Zygisk, Frida, Play Integrity API

1 UVOD

Android je danes najpopularnejši operacijski sistem za mobilne telefone z več kot 65 % tržnega deleža [9]. Sistem privzeto omeji uporabnika in aplikacije na t. i. peskovnik, v katerem nimajo skrbniških pravic (angl. *root permissions*) [37]. Uporabniki in aplikacije torej ne morejo spreminjati vseh datotek v sistemu, kar zagotavlja raven varnosti, saj imajo morebitne zlonamerne aplikacije omejen dostop do datotečnega sistema. Poleg zaščite pred zlonamerno programsko opremo peskovnik uporabniku preprečuje nenamerne spremembe v datotečnem sistemu, ki bi lahko povzročile nedelovanje operacijskega sistema. Za obhod teh omejitev mora uporabnik pridobiti skrbniški dostop, kar vključuje spremembo sistemskega delovanja naprave (angl. *rooting*).

Na napravah s skrbniškim dostopom lahko aplikacije, ki jih zaženemo kot skrbnik, s spreminjanjem zaščitenih sistemskih datotek obidejo varnostne omejitve, ki jih Android implementira z ločevanjem uporabniškega in sistemskega prostora. Posledično lahko zlonamerna programska oprema pridobi popoln nadzor nad sistemom [24]. Na napravah s skrbniškim dostopom lahko zlonamerne aplikacije spreminjajo izvorno kodo drugih aplikacij, prestrezajo in spreminjajo omrežni promet, zaobidejo preverjanje nakupov znotraj aplikacij ter prestrezajo vnose tipkovnice in datotečne vnose [37], [21]. Z naštetimi dejanji lahko zlonamerne aplikacije dobronamerne aplikacije spravijo v stanje spremenjenega delovanja oz. nedelovanja ter dostopajo do osebnih podatkov uporabnika, kot so vpisni podatki. Kljub opisanim nevarnostim ima skrbniški način Androida določene prednosti, kot so odstranjevanje nezaželenih prednaloženih aplikacij, izvedba varnostne kopije celotnega sistema in uporaba dodatnih funkcionalnosti, ki jih prednaložena različica sistema ne omogoča [25], [27].

Prejet 25. avgust, 2025
Odobren 22. december, 2025

Avtorske pravice: © 2026
Creative Commons Attribution 4.0
International License

Razvijalci aplikacij lahko zaradi navedenih nevarnosti vgradijo metode zaznave skrbniškega dostopa (angl. *root detection*), s katerimi poskušajo preprečiti izvajanje svojih aplikacij na modificiranih napravah. Skupaj z metodami zaznave skrbniškega dostopa pa se razvijajo tudi metode, ki poskušajo obiti zaznavo. Ker ima skrbniški uporabnik popoln nadzor nad sistemom, medtem ko so aplikacije privzeto omejene na peskovnik, je v teoriji vse metode zaznave mogoče obiti [16], [37]. Cilj proizvajalcev pri implementaciji zaščite je torej doseči tolikšno stopnjo kompleksnosti, da je obhod zaščite za napadalce preveč potraten.

V tem članku predstavimo metode, ki jih sodobne aplikacije uporabljajo za zaščito pred izvajanjem na napravah s skrbniškim dostopom, ter pristope za njihov obhod. Vsako aplikacijo zaženemo v navidezni napravi Android in preverimo, ali aplikacija v skrbniškem okolju preneha delovati. Nato s pregledom kode identificiramo uporabljene zaščitne mehanizme. Pri delu uporabimo aplikacije Android, ki so sestavljene iz ene ali več datotek APK, ki vključujejo izvršilno kodo v datotekah DEX (Dalvik Executable). Ker datoteka DEX ni razumljiva za branje, z obratnim prevodom (angl. *reverse engineering*) teh datotek pridobimo smali kodo, ki predstavlja zbirni jezik za Androidov izvajalnik (angl. *Android Runtime*) [30], [10]. Dobljena smali koda nam omogoča vpogled v uporabljene metode zaščite.

2 Sorodna dela

Skrbniške pravice na napravi Android lahko pridobimo na več načinov. Sun idr. [37], Nguyen-Vu idr. [25] ter Kamal idr. [14] analizirajo različne metode, s katerimi uporabniki na svojih napravah omogočijo skrbniški dostop. Skrbniški dostop lahko pridobimo tako, da zlorabimo ranljivost v jedru operacijskega sistema za namestitev binarnih datotek, ki omogočajo povečanje pravic, ali z namestitvijo spremenjene različice operacijskega sistema Android (prilagojeni ROM), kot je LineageOS [20]. Pri tem si pomagamo z orodji, kot so aplikacije z enim klikom (angl. *one-click applications*). Zhang idr. [43] analizirajo kompleksnost ranljivosti jedra Androidovega sistema, ki jih izkoriščajo tovrstne aplikacije. Aplikacije večjih proizvajalcev implementirajo naprednejše in bolj prilagojene napade na jedro Androida. Številni protivirusni programi ne zaznajo napadov aplikacij z enim klikom, kar prinaša nevarnost v primeru zlonamernih aplikacij, ki poskušajo pridobiti skrbniški dostop. Gasparis idr. [8] razvijejo orodje za zaznavo poskusa pridobitve skrbniškega dostopa v zlonamernih aplikacijah *RootExplorer*, ki uspešno identificira vse testirane vrste zlorabe ranljivosti Androida za povečanje pravic.

Za pridobitev skrbniškega dostopa se uporabljajo tudi orodja za upravljanje dostopa, kot sta zdaj nevzdrževana SuperSU [3] in Superuser. Danes priljubljeno orodje je aplikacija Magisk [39], ki omogoča brezsistemski dostop (angl. *systemless root*), pri čemer sistemske particije naprave ostanejo nespremenjene, spremeni pa se samo zagonska datoteka (angl. *boot image*). Kamal idr. [14] opisujejo uporabo Magiska in njegovih dodatnih modulov za obhod mehanizmov zaznave skrbniškega dostopa. Med opisane module spadajo Shamiko, Zygisk DenyList in MagiskHide, ki v njihovi raziskavi uspešno obidejo zaščitne mehanizme varnostno občutljivih aplikacij. Modul Zygisk omogoča dinamično instrumentacijo aplikacij s pripenjanjem na proces Zygote, ki v Androidu rojeva vse nove procese, od koder novim procesom vrine kodo modulov Magisk, ki lahko obidejo zaščito pred skrbniškim načinom. Modul Zygisk DenyList obhod zaznave stori tako, da aplikacijam na DenyListu ob rojstvu ne naloži modulov Magisk in knjižnic, ki omogočajo skrbniški dostop, zaradi česar ciljna aplikacija ne more vedeti, da imajo lahko druge aplikacije povečane pravice. Lardinois [17] nadgradi idejo Magiska in Zygisk DenyLista z rešitvijo InsecureOS, ki se bolj učinkovito skrije pred zaznavo skrbniškega načina ter omogoča prestrezanje omrežne komunikacije aplikacij.

Metode za zaznavo skrbniškega načina obravnavajo Sun idr. [37], kjer med 182 aplikacijami razlikujejo sedem glavnih kategorij zaznave. Te vključujejo preverjanje obstoja določenih paketov in datotek, preverjanje `BUILD` značk, sistemskih nastavitev, pravic direktorijev in aktivnih procesov ter poganjanje ukazov v ukazni lupini. V sklopu raziskave razvijejo orodje *RDAnalyzer*, ki aplikacijam med izvajanjem prikriva prisotnost skrbniškega dostopa s pripenjanjem na metode, ki preverjajo njegov obstoj. Tem metodam nato spreminja vhodne parametre in rezultate. Kim idr. [16] razvijejo orodje MERCIDroid, ki omogoča identifikacijo uporabljenih metod zaznave skrbniškega dostopa znotraj aplikacije. MERCIDroid s sledenjem klicev metod nariše graf toka aplikacije. Na ta način je mogoče ugotoviti povezave med preverjanjem stanja naprave in prekinitvijo izvajanja aplikacije. Preostala orodja dinamične analize, ki med drugim omogočajo obhod zaznave skrbniškega dostopa, vključujejo Frido [28] in DaVinci [5].

Orodja za dinamično analizo omogočajo opazovanje in spreminjanje stanja aplikacij med njihovim izvajanjem, kar odpravi potrebo po spreminjanju izvorne kode in ponovnem pakiranju datotek APK. S temi orodji lahko med izvajanjem aplikacije opazujemo in manipuliramo vrednosti spremenljivk, uporabniški vnos, omrežno komunikacijo ipd. V tem članku uporabljamo orodje Frida, ki nam omogoča prestrezanje klicev na ravni izvajalnika Java in nativnega vmesnika Java (JNI) ter njihovo spreminjanje, s čimer preprečimo zaznavo skrbniškega dostopa. Poleg dinamične analize z metodo obratnega prevoda predstavimo, kako s statično analizo, torej s pregledom in spreminjanjem izvorne kode, onemogočiti zaznavo skrbniškega dostopa v izbranih aplikacijah. Določene aplikacije imajo vgrajeno tudi zaznavo pripenjanja (angl. *hooking detection*), s katero

poskušajo preprečiti dinamično analizo. Stopnja zaščite se med aplikacijami razlikuje, zaradi česar univerzalna metoda za obhod zaznave skrbniškega dostopa ne obstaja [25].

Ob objavi aplikacij Android je koda lahko popačena (angl. *obfuscated*), torej spremenjena tako, da ni človeku intuitivna za branje. Dong idr. [4] razvrstijo metode pačenja kode v tri kategorije: preimenovanje identifikatorjev (angl. *identifier renaming*), šifriranje nizov (angl. *string encryption*) in uporaba Javine refleksije (angl. *Java reflection*). Rezultati njihove analize kažejo, da je 43 % aplikacij v trgovini Google Play vsaj delno popačenih.

Poleg pačenja dodatno zaščito pred statično analizo prinaša zaščita datoteke APK pred razpakiranjem (angl. *unpacking*), kar aplikacijo varuje pred branjem ali krajo njene kode. V ta namen se uporabljajo orodja, kot so LIAPP, Bangcle, Ijiami ipd. [42] Kim idr. [15] opišejo načine zaščite pred razpakiranjem aplikacij ter poleg prej naštetih orodij predstavijo lastno rešitev za zaščito. Opisana orodja zašifrirajo datoteke DEX in jih premaknejo v drugo mapo, kjer se ob zagonu aplikacije dinamično dešifrirajo in naložijo v pomnilnik. Za obhod zaščite s šifriranjem kode so razvita orodja, kot je PackerGrind, ki so ga razvili Xue idr. [40], [41]. V tem članku zaznamo uporabo orodja za zaščito kode DexProtector, katerega delovanje in obhod v kontekstu zlonamernih aplikacij analizirata Lim in Yi [19]. Načine zaščite, ki jih nudijo pakirniki DexProtector, LIAPP in DashO, v novejšem delu opiše Kalauner [13], ki za vsak način zaščite poda njen obhod in mogoče nadgradnje pakirnikov. Opiše tudi obhod zaznave skrbniškega načina zapakiranih aplikacij s priključevanjem na nativne metode (angl. *native methods*), saj je bila statična analiza popačene kode pakirnikov pretežka.

Za zaznavo izvajanja aplikacije v emulatorju Androidovega sistema lahko razvijalci uporabijo Googlov Play Integrity API [33]. Ibrahim idr. [12] v analizi uporabe njegovega predhodnika, SafetyNet Attestation API, poročajo, da ga nobena izmed pregledanih aplikacij ne implementira pravilno.

3 Metodologija

3.1 Testno okolje

Za preizkušanje obhodov zaznave skrbniškega dostopa smo v Android Studiu pripravili testno okolje v obliki navidezne naprave Android (angl. *Android Virtual Device oz. AVD*) z operacijskim sistemom Android 16. Ta različica je v času pisanja najnovejša in uporablja Android API 36. Za pridobitev skrbniških pravic smo uporabili aplikacijo Magisk v28.1, ki smo jo namestili z rootAVD skripto [23]. Za testiranje smo uporabili tri ločene navidezne naprave. Na prvi napravi, ki smo jo uporabili za testiranje statične in dinamične analize, smo uporabili privzete nastavitve Magiska z izklopljenim modulom Zygisk. Drugi dve napravi smo uporabili za testiranje učinkovitosti modulov Magisk. Na prvi smo v Magisk omogočili vgrajen modul Zygisk in njegov DenyList, na drugi napravi pa smo naložili Magisk skupaj z moduli Shamiko verzije 1.0.1, Zygisk Next verzije 1.2.9 in Play Integrity Fix.

3.2 Statična analiza

Za izvedbo statične analize smo uporabili orodji JADX-GUI [36] in Apktool [38]. Apktool omogoča razpakiranje in ponovno pakiranje datotek APK, torej pretvorbo med DEX in smali kodo, ki je človeku bolj berljiva. Za lažje iskanje zaščitnih mehanizmov smo DEX dodatno prevedli v Javo z uporabo orodja JADX-GUI. Po spremembi izvorne kode aplikacije moramo novo datoteko APK podpisati, saj Android ne omogoča namestitve nepodpisanih aplikacij [34]. V ta namen smo uporabili orodje Uber Apk Signer [6], ki avtomatizira postopek podpisovanja s pomočjo orodij *apksigner* [31] in *zipalign* [35] tako, da ustvari novo digitalno potrdilo, s katerim podpiše datoteko APK in jo pripravi za pravilno namestitev na napravo Android.

3.3 Dinamična analiza

Poleg statične analize bomo v eksperimentu uporabili dinamično analizo, s katero bomo določili, ali posamezna aplikacija preverja prisotnost skrbniškega dostopa, ter ji povečanje pravic po potrebi prikrili. Aplikacija lahko na primer vsebuje metodo, ki preveri stanje naprave. V takem primeru želimo zaznati klic te metode in vrniti oz. nastaviti določeno vrednost, ki aplikaciji pove, da naprava ni v skrbniškem načinu. To storimo tako, da v ciljno aplikacijo vnesemo lastno kodo, ki spremeni del njenega izvajanja. Enak pristop bomo uporabili za obhod zaščite pred pripenjanjem in izvajanjem aplikacije v emulatorju.

Za izvedbo dinamične analize bomo uporabili orodje Frida, ki omogoča vnos lastne JavaScript kode med izvajanjem aplikacije. Orodje Frida lahko uporabimo na dva načina. Prvi temelji na uporabi knjižnice Frida Gadget, ki ne zahteva skrbniškega dostopa, saj deluje tako, da v ciljno aplikacijo vključimo Fridino objektno datoteko (angl. *Frida object file*), ki predstavlja vstopno točko za pripenjanje na aplikacijo. Drugi način vključuje uporabo programa Frida Server, ki zahteva skrbniški način, saj omogoča pripenjanje na katerokoli aplikacijo brez spreminjanja njene kode. Ker imamo v testnem okolju omogočen skrbniški dostop, bomo v eksperimentu uporabili Frida Server.

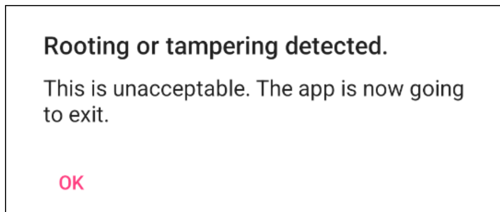
3.4 Preizkušene aplikacije

Metodi statične in dinamične analize smo uporabili za obhod zaščitnih mehanizmov v izbranih slovenskih aplikacijah. Poleg tega smo identificirali mehanizme, ki jih posamezne aplikacije uporabljajo za zaznavo skrbniškega načina in za zaznavo izvajanja v emulatorju. Pri tem smo iskali predvsem metode, ki so jih opisali Kim idr. [16]. Dodatno smo preverili učinkovitost modulov Zygisk DenyList in Shamiko kot samostojne rešitve

za obhod zaščite aplikacij. Ti moduli so v primerjavi z ročno analizo bistveno enostavnejši za uporabo, saj omogočajo prikritje skrbniškega dostopa z aktivacijo modulov v Magisku, pri čemer analiza izvorne kode aplikacij ni potrebna.

4 IZVEDBA EKSPERIMENTA

Za predstavitev metod za obhod zaščite smo uporabili javno dostopno aplikacijo UnCrackable Level 3, ki vključuje več zaščitnih mehanizmov. Pred zagonom smo pregledali njeno zaščito z orodjem APKiD [22], ki je zaznalo mehanizme za zaznavo skrbniškega načina, kot je razvidno na izpisu 1. Ob prvem zagonu aplikacije se prikaže opozorilo: *"Rooting or tampering detected,"* kot je razvidno na sliki 1, kar pomeni, da je aplikacija zaznala prisotnost skrbniškega dostopa.



Slika 1 Opozorilo, ki ga aplikacija UnCrackable Level 3 izpiše ob zagonu. Ob pritisku na gumb OK se aplikacija zapre.

Izpis 1 Izpis orodja APKiD nad aplikacijo UnCrackable Level3.

```
[*] UnCrackable-Level3.apk!classes.dex
 |-> anti_debug :
     Debug.isDebuggerConnected() check
 |-> anti_vm : Build.TAGS check
```

4.1 Statična analiza

Za začetek statične analize aplikacije UnCrackable Level 3 smo z iskalnikom v JADX-GUI poiskali ključno besedo *Rooting*, ki se izpiše na opozorilu na sliki 1, in locirali kodo, ki je prikazana na izpisu 2, s katero aplikacija preverja prisotnost skrbniškega dostopa.

Izpis 2 Rezultat orodja JADX-GUI po iskanju besede *Rooting*.

```
if (RootDetection.checkRoot1() ||
    RootDetection.checkRoot2() ||
    RootDetection.checkRoot3() ||
    IntegrityCheck.isDebuggable
        (getApplicationContext()) ||
    tampered != 0) {
        showDialog("Rooting or tampering
            detected.");
}
```

Ta del kode je prisoten v metodi `onCreate()` razreda `MainActivity`. Pogoj v tej metodi vključuje naslednje preglede:

- preverjanje rezultatov metod `checkRoot1()`, `checkRoot2()`, `checkRoot3()` in `isDebuggable()`,
- preverjanje vrednosti spremenljivke `tampered`.

Če je kateri izmed teh pogojev izpolnjen, aplikacija prikaže opozorilo in prepreči nadaljnjo uporabo. Za onemogočanje tega varnostnega mehanizma je potrebna prilagoditev ustreznega dela smali kode. Prehod na smali kodo izvedemo z razpakiranjem aplikacije z ukazom, prikazanim na izpisu 3. Nato poiščemo razred `RootDetection`, ki vsebuje metode `checkRoot1()`, `checkRoot2()` in `checkRoot3()`. Vse tri metode ročno spremenimo tako, da vedno vračajo vrednost `false`. Primer spremenjene metode je prikazan na izpisu 4, kjer je spremenjeno besedilo odebeljeno.

Izpis 3 Ukaz za razpakiranje APK-datoteke aplikacije UnCrackable Level 3.

```
apktool d UnCrackable-Level3.apk
```

Izpis 4 Spremenjena metoda `checkRoot2`, ki vedno vrača vrednost `false`. Spremenjeno besedilo je odebeljeno.

```
.method public static checkRoot2()Z
    .locals 1

    const/4 v0, 0x0
    return v0
.end method
```

Metode `isDebuggable()` razreda `IntegrityCheck` ni treba spreminjati, saj aplikacije ne gradimo z omogočenim razhroščevanjem. Za obhod preverjanja vrednosti spremenljivke `tampered` prilagodimo pogojni ukaz. Namesto ukaza:

```
if-eqz v0, :cond_1
```

zapišemo ukaz z obratnim pogojem:

```
if-nez v0, :cond_1
```

Z navedenimi spremembami zagotovimo, da se izvajanje aplikacije nadaljuje mimo varnostnega opozorila. Nato aplikacijo ponovno sestavimo z ukazom, prikazanim na izpisu 5.

Izpis 5 Ukaz za pakiranje aplikacije UnCrackable Level 3.

```
apktool b UnCrackable-Level3
```

Ob namestitvi spremenjene aplikacije v napravo dobimo napako `INSTALL_PARSE_FAILED_NO_CERTIFICATES`, saj spremenjena aplikacija ni podpisana. Postopek podpisovanja izvedemo z ukazom na izpisu 6. Po uspešnem podpisovanju aplikacijo namestimo na emulator in jo zaženemo. Mehanizem za zaznavo skrbniškega dostopa je uspešno odstranjen in aplikacija deluje brez napak.

Izpis 6 Ukaz za podpisovanje aplikacije Unrackable Level 3 z orodjem Uber Apk Signer.

```
java -jar uber-apk-signer-1.3.0.jar
-a UnCrackable-Level3.apk
```

4.2 Dinamična analiza

Za zagon programa Frida Server moramo njegovo izvorno datoteko prenesti na emulator. Datoteko prenesemo v mapo `/sdcard` z ukazom, prikazanim na izpisu 7. Direktorij `/sdcard` izberemo zato, ker gre za javno dostopni imenik, v katerem so dovoljenja za pisanje privzeto omogočena. Po prenosu datoteko premaknemo v mapo `/data/local/tmp`, saj je pogon SD-kartice priklopljen na tak način, da v mapi `/sdcard` nimamo izvršilnih pravic. Nato datoteki dodamo dovoljenje za izvajanje in jo zaženemo. Ko je Frida Server zagnan, lahko na ciljno aplikacijo pripnemo poljubno skripto Frida. Najprej poženemo prazno skripto z ukazom, prikazanim na izpisu 8. Opazimo, da se aplikacija ob zagonu samodejno zapre. Analiza skladovnega izpisa, ki ga dobimo z ukazom na izpisu 9, pokaže, da to povzroča funkcija `goodbye()`, ki je definirana v nativni knjižnici *libfoo.so*. Iz tega lahko sklepamo, da aplikacija zazna uporabo Fride in se namenoma zaključi. Del dobljenega skladovnega izpisa je viden na izpisu 10.

Izpis 7 Ukaz za naložitev datoteke Frida Serverja v mapo `/sdcard`.

```
adb push frida_server /sdcard
```

Izpis 8 Ukaz za zagon aplikacije UnCrackable Level 3, na katero je pripeta skripta Frida.

```
frida -Uf owasp.mstg.uncrackable3 -l
    script.js
```

Izpis 9 Ukaz za izpis skladovnega izpisa ob nepravilnem končanju aplikacije.

```
adb logcat --buffer=crash
```

Izpis 10 Skrajšan skladovni izpis ob končanju aplikacije UnCrackable Level 3, ko aplikacijo poženemo s prazno skripto Frida. Odebeljeno besedilo nam namiguje, da moramo spremeniti delovanje funkcije `goodbye()` v knjižnici *libfoo.so*.

```
backtrace:
#00 pc 00000000000a1ed8
    /apex/.../libc.so (tgkill+8)
    (BuildId: ...)
#01 pc 000000000000308c
    /data/app/.../lib/arm64/libfoo.so
    (goodbye()+12) (BuildId: ...)
#02 pc 00000000000031ac
    /data/app/.../lib/arm64/libfoo.so
    (BuildId: ...)
```

Ker orodji Apktool in JADX-GUI omogočata zgolj povratno prevajanje iz binarne datoteke *classes.dex* v smali jezik oz. javo, z njima ni mogoče analizirati vsebine nativnih knjižnic, ki so napisane v jeziku C in prevedene v objektno kodo. Zato za njihovo analizo uporabimo orodje Ghidra, ki omogoča povratno prevajanje objektne kode v jezik C. Z Ghidro poiščemo funkcijo `goodbye()`, v kateri sta ukaza `raise(6)` za sprožitev signala `SIGABRT` ter `_exit(0)`, ki zaključi izvajanje aplikacije, kot je razvidno iz izpisa 11.

Izpis 11 Funkcija `goodbye`, ki zaključi izvajanje aplikacije UnCrackable Level 3.

```
void goodbye(void)
{
  raise(6);
  _exit(0);
}
```

Nadaljna analiza pokaže, da funkcijo `goodbye()` kliče funkcija `FUN_001037c0`, kadar v vsebini datoteke `/proc/self/maps` zazna prisotnost nizov *"frida"* ali *"xposed"*, kot je vidno na izpisu 12. Če funkcija `FUN_001037c0` zazna katerega od omenjenih nizov, sproži končanje izvajanja prek funkcije `goodbye()`. Za obhod omenjene detekcije se z uporabo Fride pripnemo na klic funkcije `strstr()` in ji preprečimo zaznavo omenjenih nizov tako, da vrednost njenega rezultata ob izpolnjenem pogoju nadomestimo z vrednostjo `0`. Uporabljena skripta Frida je prikazana na izpisu 13.

Izpis 12 Skrajšana koda za iskanje niza "frida" med */proc/self/maps* in klic funkcije `goodbye()` ob zaznavi tega niza.

```
do {
    __stream = fopen("/proc/self/maps",
        "r");
    // ...

    pcVar1 = fgets(acStack_238, 0x200,
        __stream);
    // ...

    pcVar1 = strstr(acStack_238, "frida");
    if (pcVar1 != (char *)0x0) break;
    // ...

} while (true);

goodbye();
```

Izpis 13 Skripta Frida za obhod zaznave pripenjanja. Skripta se pripne na nativno funkcijo `strstr` in spremeni njen rezultat, če je prvi argument enak nizu *"frida"*.

```
Java.perform(() => {
    const strstr_fun =
        Module.findExportByName(null,
        "strstr");
    Interceptor.attach(strstr_fun, {
        onEnter: (args) => {
            this.frida =
                ptr(args[0]).readCString();
        },
        onLeave: (retval) => {
            if
                (this.frida.indexOf("frida")
                >= 0){
                retval.replace(0);
            }
        }
    });
```

```
});
```

Po zagonu zgornje skripte aplikacija uspešno nadaljuje izvajanje, saj ne zazna prisotnosti orodja Frida v pomnilniku naprave. Za nadaljnji obhod zaščite prestrežemo metode `checkRoot1`, `checkRoot2` in `checkRoot3` razreda `RootDetection`, ki služijo za zaznavo skrbniškega načina. Z uporabo Fride metodam spremenimo implementacijo tako, da vedno vračajo vrednost `false`. Primer skripte Frida za takšno prilagoditev je prikazan na izpisu 14. Na ta način obidemo vse obrambne mehanizme aplikacije ter preprečimo zaznavo skrbniškega dostopa in analiznih orodij.

Izpis 14 Frida skripta za obhod zaznave skrbniškega načina. Skripta se pripne na metodo `checkRoot1` razreda `RootDetection` in spremeni njen rezultat na `false`.

```
Java.perform(function() {
    const RootDetection = Java.use
    ("sg.vantagepoint.util.RootDetection");
    RootDetection.checkRoot1.implementation
        = function() {
        console.log("Bypassing the first
            Root Detection check");
        return false;
    }
});
```

5 ANALIZA SLOVENSKIH APLIKACIJ

Metode zaznave skrbniškega načina in obhode zaščite smo analizirali na 23 aplikacijah, namenjenih slovenskemu trgu. Med njimi je bilo 12 bančnih aplikacij, pet aplikacij za promet, ena aplikacija za spletno nakupovanje in pet kartic zvestobe. Med testiranimi aplikacijami smo pri desetih zaznali mehanizme zaznave skrbniškega dostopa, njihov obhod pa je bil uspešen v petih primerih, kar je razvidno v tabeli 1. Imena aplikacij so anonimizirana z namenom zaščite zasebnosti razvijalcev, opisi aplikacij pa so predstavljeni v tabeli 3. Vse aplikacije, ki so implementirale zaščito z zaznavo skrbniškega dostopa, so imele hkrati vgrajeno zaznavo izvajanja v emulatorju.

5.1 Statična analiza

Z metodo statične analize smo zaznavo skrbniškega dostopa obšli v petih testiranih aplikacijah. Preverjanje integritete aplikacije smo zaznali pri sedmih aplikacijah, pri treh izmed teh pa smo ga obšli. Za preverjanje integritete aplikacije se je v eni izmed aplikacij preverjala vrednost objekta `PackageManager` [32], ena aplikacija je preverjala obstoj določenih metod, preostale pa so uporabljale atestacijo prek komunikacije z zalednim strežnikom. Nobena izmed analiziranih aplikacij ni uporabljala atestacije Play Integrity API za preverjanje veljavnosti aplikacije.

5.2 Dinamična analiza

Obhod zaščite z uporabo dinamične analize je bil uspešen v istih aplikacijah kot obhod s statično analizo. Pri nobeni izmed aplikacij, katerih zaščito smo obšli, ni bilo prisotne zaščite z zaznavo pripenjanja. V vseh štirih aplikacijah, ki so vsebovale zaznavo pripenjanja, je bila ta prisotna v sklopu orodja DexProtector [18], pri dveh aplikacijah pa je bila poleg tega implementirana v ločenih metodah. V teh metodah se je preverjala prisotnost procesov `frida` in `xposed`. V eni aplikaciji je bil prisoten razred z imenom `AntiFridaHook`, ki je bil prazen.

5.3 Obhod z moduli Magisk

Pri štirih izmed petih aplikacij, pri katerih smo obšli zaščito, je bil njen obhod uspešen z vključitvijo modulov Magisk, neuspešen pa je bil pri obhodu zaščite aplikacije App 9. Modula Magisk Shamiko in Zygisk DenyList sta bila enako uspešna med analiziranimi aplikacijami. Vključitev modula Play Integrity Fix v nobeni aplikaciji ni imela vpliva na uspešnost obhoda, saj nobena izmed testiranih aplikacij ni implementirala Play Integrity API.

5.4 Zaznani mehanizmi in orodja za zaščito

Vse izmed testiranih aplikacij, ki implementirajo zaznavo skrbniškega dostopa, uporabljajo več mehanizmov zaznave hkrati, kar je razvidno iz tabele 2. V štirih izmed aplikacij, katerih zaščito smo obšli, so vsi ti mehanizmi uporabljeni v enem razredu, zaradi česar sta iskanje zaščite in njen obhod trivialna. Ena aplikacija, katere zaščito smo obšli, je logiko za preverjanje skrbniškega dostopa razdelila na več delov, kar je skupaj s popačeno kodo bistveno podaljšalo čas analize in onemogočilo obhod z uporabo modulov Magisk. Pri tej aplikaciji smo poleg iskanja nizov, ki nakazujejo na zaščito pred skrbniškim dostopom, opazovali tudi sledi sklada na različnih točkah izvajanja aplikacije.

Za zaznavo skrbniškega načina so aplikacije poleg lastnosti `Build.TAGS`, ki jih navajajo Sun idr. [37], pregledovale še druge lastnosti `Build` objekta, kot so `FINGERPRINT`, `MODEL`, `MANUFACTURER` ipd. Štiri aplikacije so za zaščito uporabljale knjižnico RootBeer [1], ki je dobro poznana odprtokodna knjižnica za zaznavo skrbniškega dostopa, zaradi česar je njena prisotnost lahko hitro prepoznana in obita. RootBeer je bil prisoten tako v obliki knjižnice Java kot v obliki nativne knjižnice RootBeerNative. Preostale odprtokodne rešitve, ki so bile uporabljene v aplikacijah, vključujejo Framgia Emulator Detector [7] (v eni aplikaciji) ter implementacije, povzete po odgovoru na spletnem forumu Stack Overflow [26] (v dveh aplikacijah). V dveh aplikacijah del kode za zaznavo skrbniškega načina, ki je bil vključen v obliki knjižnice, ni bil klican nikjer, temveč se je zaznava skrbniškega dostopa izvedla drugod v kodi. Tovrstno vključevanje odvečne oz. zavajajoče zaščitne kode je podaljšalo čas, potreben za iskanje dejanske zaščite. V 20 analiziranih aplikacijah

del zaznave skrbniškega načina ni onemogočal uporabe aplikacije, ampak je bil zgolj del zunanjih knjižnic za analitiko.

V vseh analiziranih aplikacijah je bil vsaj del izvorne kode popačen. V štirih primerih je bilo za ta namen uporabljeno orodje DexGuard [11]. V štirih aplikacijah je bil uporabljen pakirnik DexProtector, ki je poskuse obhoda zaščite naredil časovno neučinkovite. Ena aplikacija je imela del kode zaščitene s pakirnikom, ki ga nismo prepoznali.

6 SKLEPNE UGOTOVITVE IN ZAKLJUČEK

Uporabnik sistema Android z vklopom skrbniškega načina pridobi popoln dostop do vseh sistemskih datotek. Čeprav to seveda prinaša prednosti, pa hkrati pomeni povečano varnostno tveganje za aplikacije, saj lahko na takih napravah zlonamerna programska oprema z večjimi pravicami posega v njihovo delovanje. Da bi zmanjšale to tveganje, nekatere aplikacije implementirajo mehanizme za zaznavo skrbniškega načina, kar je najpogosteje v aplikacijah, ki obdelujejo občutljive podatke. S tem se je začela tekma med razvijalci, ki želijo svoje aplikacije čim bolje zaščititi, in uporabniki, ki te zaščite poskušajo obiti.

V tem članku smo opisali in testirali štiri metode za obhod zaščite pred skrbniškim načinom na primerih realnih aplikacij. Najprej smo predstavili statično analizo, kjer smo s spreminjanjem izvorne kode preprečili zaznavo skrbniškega načina. Nato smo predstavili dinamično analizo, pri kateri smo z uporabo orodja Frida Server spreminjali vedenje aplikacije med njenim izvajanjem. Poleg tega smo z moduloma Magisk, Zygisk DenyList in Shamiko, testirali učinkovitost modulov Magisk za prikritje skrbniškega dostopa.

Zaznava skrbniškega načina na trgu ni pogosta, njene implementacije pa so lahko zelo preproste, zaradi česar ne zagotavljajo učinkovite zaščite. Na analiziranem vzorcu 13 od 23 aplikacij ni vsebovalo mehanizmov za zaznavo skrbniškega dostopa. Zaščita je bila najpogosteje prisotna v aplikacijah iz finančnega sektorja, torej v bančnih aplikacijah. Pri polovici aplikacij z zaščito smo to obšli, pri preostalih pa je bil obhod zaradi uporabe kompleksnejših tehnik časovno neučinkovit. Uporaba orodij za šifriranje kode je bila najbolj učinkovita oblika zaščite, saj pri nobeni aplikaciji, ki je del svoje kode zašifrirala, nismo obšli zaščite. Prihodnje raziskave bi naše delo lahko razširile na kompleksnejše primere, kjer z opisanimi metodami nismo obšli zaščite. Poseben poudarek bi lahko namenili analizi pakirnikov, ki izvajajo šifriranje kode in njeno dinamično nalaganje.

V okviru analize posamezne aplikacije smo predpostavili, da aplikacija ne vsebuje zaščite pred skrbniškim načinom, če smo lahko prešli uvodne zaslone, ne da bi se aplikacija zaprla, pri nobeni aplikaciji pa nismo izvedli prijave v uporabniški račun. Ta pristop bi v prihodnosti lahko izboljšali tako, da bi preverili delovanje vseh funkcionalnosti posamezne aplikacije. V aplikacijah je namreč lahko z zaščito zavarovan le določen del, medtem ko so zasloni, ki ne obdelujejo občutljivih podatkov, kot so začetni zasloni pred prijavo, nezavarovani.

7 DISKUSIJA

7.1 Učinkovitost zaščite pred skrbniškim dostopom

Prispevek ponuja vpogled v uporabo različnih orodij za instrumentacijo Androidovih aplikacij in v trenutno stanje zaznave skrbniškega dostopa v aplikacijah, namenjenih slovenskemu trgu. Načini zaščite postajajo vse bolj napredni, kar za njihov obhod zahteva poglobljeno razumevanje delovanja sistema Android in konkretne ciljne aplikacije.

Uspeh pri obhodu zaščite je precej odvisen od tega, koliko časa, truda in finančnih sredstev razvijalci vložijo v zaščito aplikacije. Če bi imeli na voljo neomejeno časa, bi bile teoretično vse zaščite premostljive, vendar že uporaba pakirnikov in orodij za pačenje kode, ki branje in spreminjanje kode spremenijo v dolgotrajen proces, pogosto doseže dovolj visoko oviro, da se napadalcem iskanje obhoda ne izplača. Pri nekaterih analiziranih aplikacijah so že preprosti ukrepi, kot so pačenje kode, vstavljanje zavajajoče zaščitne kode in razbitje logike zaščite na več delov, bistveno podaljšali čas iskanja pravega mehanizma.

7.2 Primerjava učinkovitosti ročnih in avtomatskih metod za obhod zaščite

Dinamična in statična analiza sta bili na našem vzorcu aplikacij enako uspešni. Čeprav se za preprečevanje vsake izmed teh dveh metod uporabljajo različne vrste zaščite, velik del zaščite pred skrbniškim načinom ne izvaja zaznave pripenjanja ali preverjanja podpisa. Poleg tega sta te dve vrsti zaščite na našem vzorcu med najmanjkrat uporabljenimi, hkrati pa sta pogosto implementirani v enakem razredu kot preostala koda za zaščito. V prihodnosti bi lahko na večjem vzorcu preverili, ali obstaja razlika v pričakovani uspešnosti glede na to, ali uporabimo statično ali dinamično analizo. Zanimiv rezultat je tudi, da več kot polovica aplikacij z zaščito ni izvajala zaznave pripenjanja, čeprav so orodja za dinamično analizo, kot sta Frida in LSPosed, dobro poznana in pogosto uporabljena.

Prav tako je zanimivo, da bi pri večini aplikacij, katerih zaščito smo obšli z ročnimi metodami, obhod lahko izvedli brez podrobne analize delovanja, zgolj z uporabo preprostih orodij, kot sta Zygisk DenyList in Magisk Shamiko. Le ena izmed aplikacij je zaščito razdelila na več delov, med katerimi ni bilo očitnih povezav, kar je onemogočilo delovanje navedenih modulov Magisk in zahtevalo ročno instrumentacijo. To nakazuje, da razvijalci pogosto vključijo zaznavo skrbniškega načina v osnovni obliki, ne da bi jo dodatno izboljšali s prikrivanjem logike zaščite.

Tabela 1 Prisotnost varnostnih mehanizmov v testiranih aplikacijah. Imena aplikacij so anonimizirana z namenom zaščite zasebnosti razvijalcev.

Oznaka	Kategorija	Zaznava skrbniškega načina	Zaznava emulacije	Zaznava pripenjanja	Uspešen obhod
App 1	Finance	Da	Da	Ne	Da
App 2	Finance	Da	Da	Da	Ne
App 3	Finance	Da	Da	Da	Ne
App 4	Finance	Ne	Ne	Ne	/
App 5	Finance	Da	Da	Da	Ne
App 6	Finance	Ne	Ne	Ne	/
App 7	Finance	Da	Da	Da	Ne
App 8	Finance	Ne	Ne	Ne	/
App 9	Finance	Da	Da	Ne	Da
App 10	Finance	Da	Da	Ne	Ne
App 11	Finance	Da	Da	Ne	Da
App 12	Finance	Da	Da	Ne	Da
App 13	Prevoz	Da	Da	Ne	Da
App 14	Prevoz	Ne	Ne	Ne	/
App 15	Prevoz	Ne	Ne	Ne	/
App 16	Prevoz	Ne	Ne	Ne	/
App 17	Prevoz	Ne	Ne	Ne	/
App 18	Spletna trgovina	Ne	Ne	Ne	/
App 19	Kartica zvestobe	Ne	Ne	Ne	/
App 20	Kartica zvestobe	Ne	Ne	Ne	/
App 21	Kartica zvestobe	Ne	Ne	Ne	/
App 22	Kartica zvestobe	Ne	Ne	Ne	/
App 23	Kartica zvestobe	Ne	Ne	Ne	/

Tabela 2 Uporabljene metode za zaznavo skrbniškega načina in spreminjanja izvorne kode v posameznih aplikacijah.

Oznaka	Binarne datoteke	Aplikacije za doseganje skrbniškega načina	BUILD značke	Preverjanje podpisa	Šifriranje kode
App 1	Da	Da	Da	Da	Ne
App 2	Da	Da	Da	Da	Da
App 3	Da	Da	Da	Da	Da
App 5	Da	Da	Da	Da	Da
App 7	Da	Da	Da	Da	Da
App 9	Da	Da	Da	Da	Ne
App 10	Da	Da	Da	Da	Da
App 11	Da	Da	Da	Ne	Ne
App 12	Da	Da	Da	Ne	Ne
App 13	Da	Da	Da	Ne	Ne

Tabela 3 Opisi namenov testiranih aplikacij.

Oznaka	Opis
App 1	Bančna aplikacija slovenske banke.
App 2	Bančna aplikacija tuje banke.
App 3	Bančna aplikacija slovenske banke.
App 4	Aplikacija za sklepanje poslovnega razmerja z banko.
App 5	Elektronska denarnica slovenske banke.
App 6	Bančna aplikacija slovenske banke.
App 7	Slovenska aplikacija za denarna nakazila.
App 8	Bančna aplikacija slovenske banke.
App 9	Bančna aplikacija slovenske banke.
App 10	Bančna aplikacija tuje banke.
App 11	Bančna aplikacija slovenske banke.
App 12	Slovenska aplikacija za denarna nakazila.
App 13	Slovenska aplikacija za nakup vozovnic.
App 14	Slovenska aplikacija za izposojo prevoznih sredstev.
App 15	Slovenska aplikacija za izposojo prevoznih sredstev.
App 16	Tuja aplikacija za plačevanje parkirnine.
App 17	Slovenska aplikacija za plačilo uporabe javnega prometa.
App 18	Aplikacija slovenske spletne trgovine.
App 19	Kartica zvestobe slovenske bencinske črpalke.
App 20	Kartica zvestobe slovenske trgovine.
App 21	Kartica zvestobe slovenske trgovine.
App 22	Kartica zvestobe slovenske trgovine.
App 23	Kartica zvestobe slovenske trgovine.

7.3 Vloga emulacije

Vse analize so bile izvedene na emulatorju. Uporaba emulatorja se je izkazala za oteževalni dejavnik, saj so vse testirane aplikacije poleg zaznave skrbniškega načina izvajale tudi zaznavo okolja. Predvidevamo, da bi bila analiza na fizični napravi v številnih primerih učinkovitejša, saj bi s tem izločili potrebo po dodatnem obhodu zaznave izvajanja v navidezni napravi.

7.4 Možnosti prihodnjih raziskav

Opisane metode za obhod zaščite bi pri prihodnjih delih lahko preizkusili na novejših različicah sistema Android, ki bodo morda nudile dodatne varnostne mehanizme. Prav tako bi bilo zanimivo spremljati, katere aplikacije iz našega vzorca bodo svojo zaščito nadgradile in katere ne. Poleg tega bi lahko opazovali, ali testirane aplikacije uspešno zaznajo izvajanje s skrbniškimi pravicami, če za njihovo pridobitev uporabimo drugačna orodja, kot sta KernelSU in APatch. Metode pridobivanja skrbniškega načina in njegovega skrivanja pred aplikacijami se hitro razvijajo, zato bi v prihodnosti lahko sledili njihovemu napredku.

Dobro bi bilo tudi slediti, ali se bo v prihodnosti razširila uporaba Play Integrity API. Med raziskovanjem nismo zasledili nobene raziskave, ki bi opazovala pogostost uporabe Play Integrity API v aplikacijah. Vse dosedanje raziskave na tem področju se osredotočajo na njegovega predhodnika SafetyNet API, ki od januarja 2025 ni več podprt [29]. V prihodnosti bi takšne raziskave lahko posodobili z opazovanjem uporabe Play Integrity API. V aplikacijah, ki smo jih testirali v tem članku, Play Integrity API ni bil nikoli uporabljen, kar je morda posledica dejstva, da je njegova uporaba ob več kot 10.000 poizvedbah dnevno plačljiva. Poleg tega Play Integrity API ne zagotavlja, da je aplikacija uporabljena na nespremenjeni napravi [33].

V tem članku smo opazovali zgolj zaščito aplikacij na napravah Android. Podobno raziskavo bi bilo smiselno narediti na napravah iOS, kjer se metode za pridobitev in zaznavo skrbniškega dostopa razlikujejo od Androida. Na napravah iOS so za preverjanje stanja naprave na voljo drugačna orodja, kot je DeviceCheck storitev [2], ki deluje podobno kot Play Integrity API.

LITERATURA

[1] Scott Alexander-Bown. Rootbeer. https://github.com/scottyab/rootbeer/. Dostopano 20. maj 2025.

[2] Apple. Devicecheck. https://developer.apple.com/documentation/devicecheck. Dostopano 20. maj 2025.

[3] Jorrit Jongma (Chainfire). Supersu. https://supersuroot.org/. Dostopano 20. maj 2025.

[4] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu, editors, *Security and Privacy in Communication Networks*, pages 172–192, Cham, 2018. Springer International Publishing.

[5] Alexander Druffel and Kris Heid. Davinci: Android app analysis beyond frida via dynamic system call instrumentation. In Jianying Zhou, Mauro Conti, Chuadhry Mujeeb Ahmed, Man Ho Au, Lejla Batina, Zhou Li, Jingqiang Lin, Eleonora Losiouk, Bo Luo, Suryadipta Majumdar, Weizhi Meng, Martín Ochoa, Stjepan Picek, Georgios Portokalidis, Cong Wang, and Kehuan Zhang, editors, *Applied Cryptography and Network Security Workshops*, pages 473–489, Cham, 2020. Springer International Publishing.

[6] Patrick Favre-Bulle. Uber apk signer. https://github.com/patrickfav/uber-apk-signer. Dostopano 20. maj 2025.

[7] Framgia. Framgia emulator detector. https://github.com/framgia/android-emulator-detector. Dostopano 20. maj 2025.

[8] Ioannis Gasparis, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. Detecting android root exploits by learning from root providers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1129–1144, Vancouver, BC, August 2017. USENIX Association.

[9] Statcounter GlobalStats. Mobile operating system market share worldwide. Dostopano 20. maj 2025.

[10] Google. Smali. https://github.com/google/smali. Dostopano 20. maj 2025.

[11] Guardsquare. Dexguard. https://www.guardsquare.com/dexguard. Dostopano 20. maj 2025.

[12] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. Safetynot: on the usage of the safetynet attestation api in android. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’21, page

150–162, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Paul Gabriel Kalauner. Analysis and bypass of android application anti-reverse engineering mechanisms. *TU Wien*, 2023.

[14] Bilal Ahmed Kamal, Abdul Basit Shahid, Syed Muhammad Sajjad, Kashif Kifayat, and Khwaja Mansoor Ul Hassan. A novel technique of android stealth rooting. In *2024 17th International Conference on Development in eSystem Engineering (DeSE)*, pages 275–280, 2024.

[15] Nak Young Kim, Jaewoo Shim, Seong-je Cho, Minkyu Park, and Sangchul Han. Android application protection against static reverse engineering based on multidexing. *J. Internet Serv. Inf. Secur.*, 6(4):54–64, 2016.

[16] Taehun Kim, Hyeonmin Ha, Seoyoon Choi, Jaeyeon Jung, and Byung-Gon Chun. Breaking ad-hoc runtime integrity protection mechanisms in android financial apps. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 179–192, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Simon Lardinois and Axel Legay. Modifying android for security analysis. Master's thesis, Ecole polytechnique de Louvain, Université catholique de Louvain, 2023.

[18] Licel. Dexprotector. https://dexprotector.com/. Dostopano 20. maj 2025.

[19] Jongsu Lim and Jeong Hyun Yi. Structural analysis of packing schemes for extracting hidden codes in mobile malware. *EURASIP Journal on Wireless Communications and Networking*, 2016(1):221, 2016.

[20] LineageOS. Lineageos – lineageos android distribution. https://lineageos.org/. Dostopano 20. maj 2025.

[21] Collin Mulliner, William Robertson, and Engin Kirda. Virtualswindle: an automated attack against in-app billing on android. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 459–470, New York, NY, USA, 2014. Association for Computing Machinery.

[22] Red Naga. Apkid. https://github.com/rednaga/APKiD. Dostopano 20. maj 2025.

[23] newbit. rootavd. https://gitlab.com/newbit/rootAVD/. Dostopano 20. maj 2025.

[24] Long Nguyen Vu, Ngoc-Tu Chau, Seongeun Kang, and Souhwan Jung. Droidsecure: A technique to mitigate privilege escalation in android application. *Journal of the Korea Institute of Information Security & Cryptology*, 26(1):169–176, 2016.

[25] Long Nguyen Vu, Ngoc-Tu Chau, Seongeun Kang, and Souhwan Jung. Android rooting: An arms race between evasion and detection. *Security and Communication Networks*, 2017:1–13, 10 2017.

[26] Stack Overflow. Determine if running on a rooted device. https://stackoverflow.com/a/8097801/8549541. Dostopano 20. maj 2025.

[27] Nick Rahimi, John Nolen, and Bidyut Gupta. Android security and its rooting—a possible improvement of its security architecture. *Journal of Information Security*, 10(2):91–102, 2019.

[28] Ole André V. Ravnås. Frida. https://github.com/frida/frida. Dostopano 20. maj 2025.

[29] Android razvijalci. About the safetynet attestation api deprecation. https://developer.android.com/privacy-and-security/safetynet/deprecation-timeline. Dostopano 20. maj 2025.

[30] Android razvijalci. Android runtime and dalvik. https://source.android.com/docs/core/runtime. Dostopano 20. maj 2025.

[31] Android razvijalci. Apksigner. https://developer.android.com/tools/apksigner. Dostopano 20. maj 2025.

[32] Android razvijalci. Packagemanager. https://developer.android.com/reference/android/content/pm/PackageManager. Dostopano 20. maj 2025.

[33] Android razvijalci. Play integrity api. https://developer.android.com/google/play/integrity. Dostopano 20. maj 2025.

[34] Android razvijalci. Sign your app. https://developer.android.com/studio/publish/app-signing. Dostopano 20. maj 2025.

[35] Android razvijalci. Zipalign. https://developer.android.com/tools/zipalign. Dostopano 20. maj 2025.

[36] skylot. Jadx. https://github.com/skylot/jadx. Dostopano 20. maj 2025.

[37] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. Android rooting: Methods, detection, and evasion. *SPSM '15: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2015.

[38] Connor Tumbleson. Apktool. https://apktool.org/. Dostopano 20. maj 2025.

[39] John Wu. Magisk: The magic mask for android. https://github.com/topjohnwu/Magisk. Dostopano 20. maj 2025.

[40] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 358–369, 2017.

[41] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. Packergrind: An adaptive unpacking system for android apps. *IEEE Transactions on Software Engineering*, 48(2):551–570, 2022.

[42] Rowland Yu. Android packers: facing the challenges, building solutions. In *Proceedings of the 24th Virus Bulletin International Conference*, 2014.

[43] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1093–1104, New York, NY, USA, 2015. Association for Computing Machinery.

Tim Thuma je študent na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Zanimajo ga področja razvoja programske opreme, vgrajenih sistemov in umetne inteligence. Njegovi raziskovalni interesi zajemajo področji nizkonivojske analize sistemov in varnosti programske opreme.

Tilen Medved je študent na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Zanimajo ga področja robotike, umetne inteligence in kibernetske varnosti. Njegovi raziskovalni interesi zajemajo področji zanesljivosti in varnosti programske opreme.

Matevž Pesek je izredni profesor in raziskovalec na Fakulteti za računalništvo in informatiko Univerze v Ljubljani, kjer je diplomiral (2012) in doktoriral (2018). Od leta 2009 je član Laboratorija za računalniško grafiko in multimedije. Od leta 2024 izvaja predmeta varnost programov in varnost sistemov, kjer se raziskovalno ukvarja s poučevanjem konceptov in organizacijo dogodkov s področja računalniške varnosti.