Attacco ad Android: obiettivi, rischi e contromisure

Alessandro Menti, Mattia Zago

Sommario

Le ultime tendenze del mercato mostrano come smartphone e tablet stiano subendo una rapida ed incontrollata crescita rispetto a quella dei tradizionali desktop e notebook. L'aumento delle capacità di calcolo nei device senza sacrificarne la mobilità, unita ad una presenza pervasiva degli accessi alla rete, è alla base di una vera e propria rivoluzione dello stile di vita.

Rispetto ai classici cellulari dell'ultimo ventennio uno smartphone ha la capacità intrinseca di eseguire applicazioni di terze parti che offrono funzioni che spaziano dal *social* all'ambito finanziario; non mancano le soluzioni di livello enterprise per le grandi aziende.

I moderni device sono diventati pertanto una fonte di dati personali che devono essere protetti. Lo sviluppo delle nuove versioni dei SO mobile deve dunque porre come specifici principi la tutela della privacy dell'utente e la difesa dal malware.

Questa relazione si pone come obbiettivo quello di fornire una panoramica generale dei principali meccanismi di protezione attualmente implementati, focalizzandoci in particolar modo sulla piattaforma Android. In questo contesto vedremo quali dati sensibili vengono troppo spesso resi accessibili e quali permessi sono maggiormente richiesti dalle applicazioni contenenti malware; a tal scopo analizzeremo il funzionamento di Bouncer (sistema di analisi statica e dinamica introdotto da Google) e presenteremo il caso di Kakao Talk, un'applicazione di messaggistica in voga tra i dissidenti tibetani che è stata oggetto di attacco. Nella seconda parte analizzeremo il *Security Enhanced Android Framework* esaminando i suoi principi di funzionamento, le tecniche di implementazione, una breve analisi delle performance e i vantaggi e svantaggi di questo meccanismo. Vedremo inoltre come l'analisi dinamica fornita dal framework CopperDroid possa fornire informazioni aggiuntive per il rilevamento di codice malevolo.

1 La diffusione di Android e una panoramica sul malware

Fino a pochi anni fa Symbian era leader indiscusso del mercato mobile, e pur essendo stati isolati dei malware specifici fin dal biennio 2004 – 2006 non si sono mai registrati scenari di particolare gravità in cui specifiche minacce (o loro famiglie) abbiano avuto una diffusione tale da compromettere milioni di cellulari [21]. Da quando Nokia ha iniziato a perdere quote di mercato fino ad oggi non vi è mai stato un dispositivo dotato di tale sistema operativo che predominasse nel panorama mondiale, per cui l'interesse dei malintenzionati per tale piattaforma è gradualmente scemato.

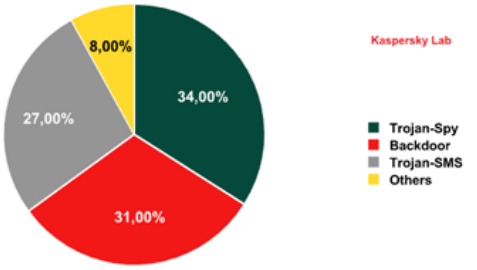

FIGURA 1. Classificazione del malware Android in base al comportamento, da [22].

iOS e BlackBerry hanno, al contrario, guadagnato numerosi punti percentuali in termini di diffusione; una crescita straordinaria è stata però registrata da Android, sistema operativo aperto che ha raggiunto una diffusione globalmente maggiore del 60% [17]. In conseguenza di ciò, per i dispositivi basati su tale piattaforma sono stati scritti numerosi nuovi *exploit*.

Nel 2012 si è assistito ad una diffusione sempre più pervasiva dei malware per Android. Secondo gli analisti di Kaspersky Lab [21, 22, 23] il rapporto fra malware per Android e minacce per altri sistemi mobile è giunto all'impressionante valore di 9:1; ciò è stato facilitato anche dall'ampia superficie di attacco offerta dal sistema operativo. L'obbiettivo principale è il furto di dati personali tramite trojan e backdoor, nonché la sottrazione di credito telefonico per mezzo di invio di SMS a numeri *premium*.

2 I fattori di rischio

2.1 La frammentazione e i suoi effetti

Il principale problema derivante dalla notevole diffusione di Android è la cosiddetta *frammentazione*. Con tale termine si indica il fenomeno della diffusione di tale sistema operativo, spesso in versioni non recenti, su tipologie di hardware prodotte da diverse aziende e/o differenti per fattore di forma, chipset e caratteristiche del dispositivo.

Un'analisi statistica del progetto OpenSignal [26], condotta da Febbraio ad Agosto 2012 su un campione di circa 700000 dispositivi che scaricavano tale applicazione, ha consentito di rilevare l'esistenza di 3997 modelli distinti e di 599 produttori di *device*. Se dal punto di vista di questi ultimi si può evidenziare la posizione dominante di pochi leader di grandi dimensioni (Samsung, HTC, SEMC, Motorola) che complessivamente coprono più della metà del mercato, la varietà dei dispositivi in circolazione è notevolmente più elevata: il modello di smartphone risultato essere a maggior diffusione — il Samsung Galaxy S II — è responsabile solo del 20% circa dei contatti, e numerosi modelli meno popolari sono stati registrati una sola vol-

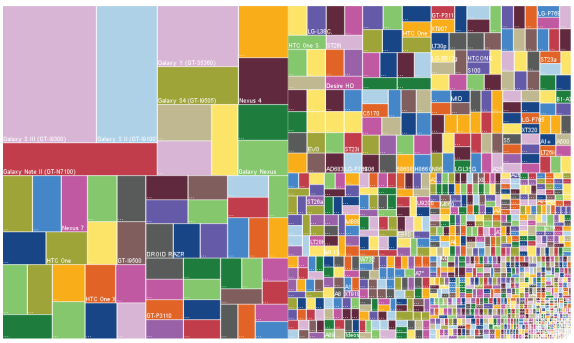

FIGURA 2. Rappresentazione grafica della frammentazione dell'ecosistema Android a luglio 2013, da [25]. Ogni tassello rappresenta un singolo modello di dispositivo.

ta. Ulteriori analisi sulla versione del sistema operativo adottato hanno rivelato che, ad Aprile 2012, il 55,4% di tali dispositivi adottava la versione 2.3.3 "Gingerbread" di Android, il 20,5% adottava la versione 2.2 "Froyo", mentre solo l'8,5% adottava la versione 4.0.3–4.0.4 "Ice Cream Sandwich" (l'ultima rilasciata all'inizio dello studio)[1].

Indagini più recenti (Luglio–Agosto 2013) condotte dallo stesso progetto e, indipendentemente, da Google mostrano mutamenti abbastanza rilevanti: si registra una prevalenza delle versioni 4.1–4.2 "Jelly Bean", installate sul 40% circa dei dispositivi, la versione "Ice Cream Sandwich" sul 22,5% e la versione "Gingerbread" sul 33% circa. Le caratteristiche generali della distribuzione dei modelli sul mercato rimane sostanzialmente analoga [11, 25].

Se da un lato si può affermare che la volontà degli utenti di disporre di uno smartphone aggiornato, o aggiornabile, alle ultime versioni del sistema operativo (per usufruire delle continue novità) ha contribuito all'inizio di un'inversione di tendenza, è altrettanto vero che una percentuale ancora molto elevata di dispositivi monta versioni vecchie (o comunque non aggiornate) dell'OS. Ciò accade nonostante:

- il codice sorgente di Android sia liberamente disponibile [14];
- esista, e sia fornita con tale codice, l'*Android Compatibility Test Suite*, volta ad automatizzare il più possibile la verifica di buon funzionamento di una ROM sul relativo dispositivo [10];
- gli sviluppatori aggiungano nelle nuove *release* del sistema operativo funzionalità di sicurezza la cui disponibilità sui terminali già in commercio apporterebbe benefici notevoli; alcuni esempi sono l'Address Space Layout Randomization (protezione contro il buffer overflow), un'API per la gestione sicura delle credenziali, SELinux (sistema per il mandatory access control) [12, 13].

I fattori che contribuiscono al verificarsi di tale fenomeno sono i seguenti.

Tempi/costi per sviluppare driver e modifiche elevati. La disponibilità del codice sorgente di Android elimina l'onere da parte dei singoli produttori di implementare nuovamente il software deputato a gestire le principali funzionalità del telefono e il sistema operativo; spesso, tuttavia, le informazioni relative ai chipset utilizzati in ambito hardware sono rese disponibili esclusivamente ai produttori stessi. Come conseguenza, questi ultimi sono obbligati a sviluppare (o personalizzare, se il produttore del chipset ha già fornito parte del codice necessario) i driver che consentono l'interazione con il kernel. Il produttore può inoltre personalizzare il dispositivo in vari modi (si va dall'introdurre programmi di utilità aggiuntivi alla sostituzione completa dell'interfaccia principale di Android, detta *launcher*). In entrambi i casi sono richiesti tempi e costi medi o elevati, da allocare nuovamente ad ogni aggiornamento sostanziale del sistema operativo (come può essere una nuova versione di Android), difficilmente giustificabili se non per terminali di fascia alta.

Tempi di test/certificazione elevati. A seguito della realizzazione di un nuovo dispositivo o di un aggiornamento, quest'ultimo deve essere testato e certificato per verificare l'assenza di bug critici e, se c'è la possibilità che l'aggiornamento influisca sulle caratteristiche elettriche o elettromagnetiche del dispositivo, accertare la conformità alle direttive internazionali (CE, FCC, direttive sulle emissioni radio). Ancora una volta, i tempi e i costi richiesti non sono ridotti.

Immissione costante sul mercato di nuovi modelli. È preferibile, per mantenere un'elevata competitività, introdurre costantemente sul mercato modelli nuovi (più potenti sotto il profilo hardware) piuttosto che aggiornare quelli vecchi.

Personalizzazione delle ROM da parte degli operatori. Alcuni operatori telefonici personalizzano ulteriormente le ROM (ad esempio inserendovi il proprio logo, imponendo il *SIM locking* o aggiungendo ulteriori utilità). Si rende quindi necessaria (per ogni aggiornamento) una ulteriore fase di test: oltre alla prima effettuata da parte del produttore del telefono, se ne deve aggiungere una seconda da parte dell'operatore (il quale può approvare o meno, a propria esclusiva discrezione, il rilascio dell'aggiornamento).

Questo processo di lenta diffusione si applica anche alle patch di sicurezza. Nel 2011 un gruppo di ricercatori di Lookout ha misurato il tempo intercorrente fra la pubblicazione della patch ufficiale da parte di Google per alcuni exploit Android e l'installazione degli aggiornamenti sulla metà del parco dispositivi coinvolto. I tempi medi nei casi presi in esame sono variati da trenta a quaranta settimane; per alcune minacce determinati operatori non hanno nemmeno rilasciato una correzione [20][2].

[1] Durante lo studio, il 9 Luglio 2012, è stato rilasciato Android 4.1 "Jelly Bean"; qui lo si è volutamente ignorato perché si è ritenuto che un mese non sia un lasso di tempo sufficiente per consentirne l'adozione su vasta scala.

[2] Una possibile soluzione a questo problema sarebbe imporre a ogni produttore — ad esempio, come clausola aggiuntiva nel contratto che consente a ogni *vendor* l'accesso da parte dei propri dispositivi allo store

2.2 Rooting

Un altro fattore di rischio di particolare importanza sui terminali Android[3] è costituito dal *rooting*, ossia dall'acquisizione di privilegi amministrativi (normalmente riservati al produttore) sul dispositivo, spesso effettuata tramite veri e propri exploit. Il rooting consente:

- l'eradicamento dalla ROM originale di *bloatware* (software promozionale e/o di dubbia utilità preinstallato dal produttore);
- la regolazione fine del comportamento dell'hardware (frequenza della CPU, voltaggi) per ottenere prestazioni migliori o risparmiare maggiore energia [30];
- l'installazione di ROM *custom* per personalizzare l'aspetto del telefono e introdurre migliorie e nuove funzionalità non presenti nelle versioni standard di Android [5];
- il miglioramento della sicurezza tramite firewall.

Di contro, l'ottenimento dei privilegi amministrativi comporta che tutte le applicazioni installate ne dispongano, rendendo molto più facile per un eventuale malware assumere il completo controllo del dispositivo.

3 Permessi

Un punto cruciale da tenere in considerazione è tuttavia il seguente: non è realmente necessario scrivere un malware che sfrutti vulnerabilità del dispositivo target per ottenere i dati sensibili di un utente. È sufficiente creare un'applicazione che richieda i permessi appropriati per accedervi.

3.1 IPC e controllo degli accessi

Per esaminare nel dettaglio i problemi relativi ai permessi è necessario innanzitutto esaminare le modalità di comunicazione fra processi (di sistema e non) in Android [2, 4].

Il modello adottato per la gestione dei permessi sui file è derivato da quello di UNIX e prevede l'esecuzione di ogni applicazione con un UID e un GID dedicati ed univoci [16]. In tal modo, impostando i permessi sui file in modalità di sola lettura (`640` o `660`), nessuna app può agire sui dati delle altre (*sandboxing*).

Ogni app può inoltre fornire uno o più dei seguenti componenti:

Activity: singola operazione eseguibile dall'utente, spesso dotata di una propria interfaccia.

Service: operazione eseguita in background, priva di interfaccia utente.

Broadcast receiver: componente deputato alla ricezione di notifiche di eventi da parte del sistema operativo (quali la ricezione di un SMS o il cambio di data e ora).

Content provider: componente deputato al salvataggio e recupero di dati, solitamente tramite l'utilizzo di un backend SQLite.

Un reference monitor regola sia le comunicazioni fra tali componenti all'interno di una singola app (ICC), sia le interazioni (*intent*) tra un'app e il sistema (IPC tramite protocollo Binder): ogni componente dispone di una *access permission label*, e ogni applicazione all'atto dell'installazione richiede un elenco di permission label necessarie. Confrontando le etichette di accesso ai componenti e alle API scritte in un file manifesto (`AndroidManifest.xml`) con le comunicazioni effettuate il reference monitor concede o nega l'accesso.

Viene inoltre eseguito un controllo sulla proprietà dei componenti stessi per evitare che un'applicazione si spacci per un'altra. Quando un processo A effettua una chiamata tramite protocollo Binder esso apre una comunicazione con il kernel, comunicando i dati da trasferire e un riferimento all'oggetto ricevente (*binder object*) del processo ricevente B. Questo passaggio, effettuato tramite il metodo `Parcel.readStrongBinder()`, fornisce la garanzia (data dal kernel) che il processo scrivente possieda effettivamente il riferimento inviato. Viene poi creato un nuovo thread per evadere la richiesta, parametrizzandolo con l'UID dell'applicazione ed il PID del processo chiamante (tali dati vengono memorizzati nello spazio del kernel). Per controllare se il chiamante ha i permessi necessari per effettuare tale richiesta il ricevente può invocare le funzioni `checkCallingPermission(String permission)` e `checkCallingPermissionOrSelf(String permission)`. In ogni momento il driver del protocollo Binder può recuperare i dati del chiamante tramite i metodi `getCallingUid()` e `getCallingPid()`.

Un punto debole del meccanismo è che l'accettazione dei permessi richiesti dall'applicazione durante l'installazione fa esclusivo affidamento sulle conoscenze dell'utente, l'"anello debole" della catena (è poco realistico assumere che un utente medio sia in grado di discriminare fra le autorizzazioni effettivamente necessarie e quelle che non lo sono).

Tralasciando bug e permessi sui file settati erroneamente a world-readable [3] il problema principale è quello di analizzare i principali permessi necessari alle applicazioni e le loro finalità [15]. Spesso viene richiesto l'accesso agli identificatori del device, in particolare quelli del gruppo `PERSONAL_INFO` e quelli atomici `READ_PROFILE` e `READ_PHONE_STATE` che vengono utilizzati come *fingerprint* dell'utente. I permessi più pericolosi però sono altri, e spaziano dall'utilizzo di sistemi che prevedono un costo a quelli che consentono il blocco definitivo del device:

ufficiale [9] — il rilascio di aggiornamenti di sicurezza entro un breve tempo dal relativo annuncio. La pressione esercitata dai consumatori per poter accedere allo store ufficiale — almeno nel mondo occidentale (in Oriente sono popolari store alternativi) — dovrebbe costituire un fattore critico per spingere le case produttrici ad accettare. Non è noto se tale misura sia stata implementata.

[3] Il fenomeno è presente anche su smartphone dotati di altri sistemi operativi, ma assume maggior rilevanza nel mondo Android in quanto la natura aperta del sistema operativo e l'indipendenza dalla piattaforma dei relativi strumenti di sviluppo ha reso più facile le personalizzazioni che qui si espongono.

ACCESS_*_LOCATION: i permessi di questa famiglia consentono di geolocalizzare il dispositivo in maniera via via più precisa.

BRICK: consente di disabilitare permanentemente il device.

CALL_PRIVILEGED: consente all'applicazione di effettuare ogni genere di chiamata senza che l'utente ne sia al corrente.

DUMP: consente di effettuare un dump completo delle informazioni di sistema.

INJECT_EVENTS: consente di generare eventi (pressioni dei tasti, tocchi sullo schermo) e di inserirli nel gestore degli eventi di qualsiasi altra applicazione o finestra.

READ_SMS: consente di leggere gli SMS.

SEND_SMS: consente di inviare SMS senza conferma da parte dell'utente.

WRITE_SECURE_SETTINGS: consente di modificare la policy di sicurezza (requisiti di robustezza della password) del sistema.

3.2 Il caso Kakao Talk

Effettuare il repack di un'applicazione famosa per poi offrirla gratis sullo store ufficiale è il metodo ad oggi più diffuso per rubare dati personali[4]. La procedura generalmente seguita è quella di copiare il codice dell'applicazione aggiungendovi parti di codice che richiedono permessi aggiuntivi al fine di ottenere in maniera *lecita* le informazioni personali e/o il controllo del device.

Questo è stato il caso dell'applicazione Kakao Talk, un client di messaggistica istantanea in voga nella comunità tibetana che è stato oggetto di un attacco volto a manipolare l'app presente sul dispositivo di una delle personalità più importanti per tracciarne posizione, messaggi e contatti [32].

Il vettore di infezione nel caso in esame è stato un messaggio di posta elettronica contenente il pacchetto infetto come allegato (un blocco di rete impediva l'accesso allo store ufficiale, motivo per cui l'invio di pacchetti `.apk` per e-mail era usuale e assolutamente non sospetto). L'analisi ha rivelato l'aggiunta di permessi e procedure malevoli operata senza intaccare il nucleo funzionale e l'interfaccia dell'applicazione, nonché l'aggiunta di una firma digitale con un certificato falsificato contenente dati casuali.

Il malware richiedeva una notevole quantità di permessi aggiuntivi per accedere agli account memorizzati sul telefono, ai messaggi, alla geolocalizzazione, al file system, alle chiamate, al Bluetooth e alla configurazione del dispositivo. In Figura 3 sono evidenziati tali permessi.

Più in dettaglio, l'applicazione memorizzava a intervalli regolari e in forma criptata la rubrica dell'utente, la cronologia degli SMS e delle chiamate e le informazioni sulla rete telefonica in un file. Dopo aver contattato un server Command and Control, essa scaricava informazioni

[4] Un *case study* al riguardo può essere consultato in [6].

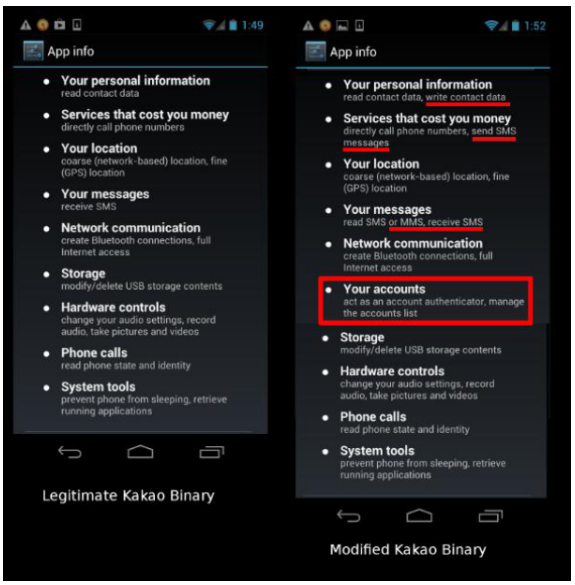

FIGURA 3. Confronto tra i permessi richiesti dalla versione di Kakao Talk originale e da quella modificata, da [32]. In rosso sono evidenziati i permessi aggiuntivi necessari per il malware.

di configurazione come URL e credenziali necessarie per l'upload del file incriminato. La connessione avveniva visitando un sito con grafica analoga a quella di Baidu (un noto motore di ricerca cinese) che nel sorgente HTML conteneva il codice di configurazione per il malware sotto forma di commenti. Essa infine intercettava gli SMS contenenti uno specifico codice; se presente, veniva inviato un SMS di risposta al cui interno erano riportati i dati di rete sufficienti per geolocalizzare in maniera precisa il device.

Alcuni dei permessi richiesti sembrano però non essere stati mai sfruttati, come ad esempio l'utilizzo del GPS, l'accesso Bluetooth e l'avvio dell'applicazione all'accensione del dispositivo, lasciando presumere che l'attaccante abbia pianificato una possibile evoluzione del software.

Il file modificato non è stato rilevato come malware da nessuno degli antivirus utilizzati (Avast, Lookout, Kaspersky) ed è stato necessario un intervento umano per capire il tipo di modifiche operate sull'applicazione.

4 Bouncer: un tentativo di eliminare il malware

4.1 Funzionamento

In risposta alla crescita della diffusione di malware sulla piattaforma Android, e in particolare sul relativo store, Google ha introdotto a Febbraio 2012 uno strumento di scansione automatica delle applicazioni pubblicate, denominato *Bouncer*. In base all'annuncio ufficiale [19], l'obiettivo di tale servizio è quello di prevenire il caricamento su Google Play Store di applicazioni malevole tramite tre tecniche:

1. analisi statica delle applicazioni per ricercare minacce note;

2. esecuzione del software in un emulatore virtuale, con monitoraggio e registrazione di comportamenti sospetti, seguito da una revisione manuale da parte di analisti esperti nel caso in cui l'accertamento abbia dato esito positivo;

3. analisi dei dati immessi in fase di registrazione dai nuovi sviluppatori, per prevenire iscrizioni ripetute e/o con dati falsi da parte degli autori di malware.

Bouncer, nelle intenzioni di Google, dovrebbe costituire la principale linea di difesa contro il codice malevolo, unitamente al sandboxing e alla corretta gestione dei permessi, in quanto:

- non è né ammissibile né ragionevole ritenere che l'utente medio sia sempre in grado di discriminare le applicazioni malevole da quelle che non lo sono, e di assumersi la relativa responsabilità;
- sebbene vi sia da parte di Google la possibilità di disinstallare dai dispositivi eventuali applicazioni malevole installate dallo store, senza intervento dell'utente, tale misura agisce esclusivamente *a posteriori*, quando il malware ha già avuto l'opportunità di causare un danno;
- per consentire una pubblicazione più rapida da parte degli sviluppatori e a causa del grande numero di applicazioni presenti, non si procede a una revisione manuale di ogni applicazione pubblicata, al contrario di quanto è previsto per altri store (Apple Store, BlackBerry World).

4.2 Tecniche di evasione

A causa della scarsità di dettagli riportati nell'annuncio e del cruciale ruolo preventivo svolto, Bouncer si è classificato in tempi brevi come un obiettivo primario per i ricercatori nell'ambito della sicurezza. Pochi mesi dopo la messa a regime del sistema sono state pubblicate due analisi volte a determinare le caratteristiche tecniche del motore di scansione e a tentare di aggirare lo stesso.

La prima, eseguita da Jon Oberheide e Charlie Miller [24], è stata in grado di fornire una panoramica completa del sistema. I ricercatori hanno provveduto a registrare più account per sviluppatori Google Play e ad inviare più versioni di un'applicazione (appositamente realizzata) in grado di raccogliere i dati del dispositivo su cui viene eseguita e di trasmetterli a un server controllato da essi. Da tale esperienza si è riuscito a inferire quanto segue.

- Bouncer analizza ogni applicazione inviata pochi minuti dopo l'invio della richiesta di pubblicazione e prima della pubblicazione effettiva.
- Viene operata una prima analisi statica del file `.apk` alla ricerca di pattern sospetti presenti testualmente nel pacchetto stesso (un esempio in merito è `/system/bin`, directory dei binari di sistema, l'analogo di `/bin` nei sistemi UNIX-like).
- Se il pacchetto supera l'analisi statica, viene operata un'analisi dinamica eseguendo l'applicazione per cinque minuti circa in un emulatore basato su QEMU (gli identificativi del dispositivo sono tuttavia alterati in modo tale che il software testato sia indotto a credere di essere eseguito su un dispositivo reale).
- Per coprire il maggior numero di casi d'uso possibile (e, di conseguenza, testare quanto più codice possibile) Bouncer simula l'interazione con l'interfaccia utente dell'applicazione (tramite tap casuali).
- L'emulatore ha un account Google associato (analogamente a quanto viene offerto in un dispositivo reale) il cui indirizzo è `miles.karlson@gmail.com`.
- Nella rubrica di sistema sono stati inseriti dei contatti[5] e sono presenti dei file all'interno delle memorie di massa (foto e file di testo, alcuni dei quali con nomi di probabile interesse per un'applicazione malevola).[6]
- L'infrastruttura di Bouncer consente l'accesso delle applicazioni testate ad Internet da blocchi di rete prefissati (`74.125.0.0/16` e `209.85.128.0/17`); è stato anche identificato il blocco di rete usato per la revisione manuale delle applicazioni (`173.194.99.0/16`).

Un approccio mirato maggiormente all'evasione di tale architettura è dato dall'analisi di Percoco e Schulte [29], in cui gli autori hanno realizzato un'applicazione legittima (per il blocco degli SMS indesiderati) destinata a contenere malware. Rilevato il blocco di rete da cui Bouncer operava, essi hanno aggiunto un semplice controllo per disabilitare eventuali funzionalità malevole nel caso in cui l'indirizzo IP corrente appartenesse allo spazio utilizzato da Bouncer. Per evitare il rilevamento manuale delle funzioni dannose nel caso in cui fosse stata eseguita una revisione del codice da parte di un operatore umano, gli autori hanno adottato una tecnica impiegata da molte applicazioni popolari per aggiornare le proprie funzionalità senza dover pubblicare nuove revisioni dell'applicazione: codificare alcune delle funzionalità in JavaScript e far sì che il programma caricasse dinamicamente tale codice da un server su Internet. Tale schema, analogo a quello impiegato dalle moderne botnet, consente una personalizzazione elevata, un facile aggiornamento e una bassa detection rate.

Terminata la realizzazione della struttura iniziale dell'app, i ricercatori hanno aggiunto in fasi successive funzionalità benigne (che richiedevano ulteriori permessi oltre a quelli concessi nella prima versione) e funzionalità malevole che sfruttavano i medesimi permessi. In tal modo la richiesta di autorizzazioni aggiuntive sarebbe stata giustificata, anche per un utente esperto, e un eventuale revisore avrebbe dovuto operare un'analisi approfondita per stabilire se il nuovo codice fosse stato in parte maligno o meno. Le funzionalità malevole venivano eseguite ogni quindici minuti.

[5] Un numero di telefono della Pennsylvania e uno corrispondente a quello della Casa Bianca.

[6] Oberheide e Miller hanno rilevato tre immagini — due casuali nella cartella `download`, una foto in `DCIM/Camera` e un file `android/data/passwords.txt`, queste ultime probabilmente *booby trap* monitorate da Bouncer; Percoco e Schulte non hanno invece rilevato fotografie. Non è ancora noto se tali file, in un determinato momento, siano presenti allo stesso modo su tutte le istanze dell'emulatore o se essi vengano modificati da un emulatore all'altro per evitare il rilevamento.

La singola scansione effettuata da Bouncer dopo (quasi) ogni aggiornamento non è stata in grado di rilevare la presenza del malware. Una seconda scansione rilevata a distanza di alcune settimane dalla pubblicazione è parimenti risultata infruttuosa.

Constatato che l'*engine* di analisi non era riuscito a rilevare la presenza delle istruzioni malevole staticamente, i ricercatori hanno rimosso le limitazioni sull'indirizzo IP del dispositivo; tale accorgimento avrebbe dovuto causare la rilevazione dinamica del malware, ma così non è accaduto. Solo la riduzione dell'intervallo fra le singole esecuzioni da quindici a un secondo ha provocato un numero massiccio di scansioni automatiche seguite da una revisione manuale e dalla disattivazione dell'account da sviluppatore dei ricercatori.

Le due esperienze hanno messo in evidenza i seguenti limiti del motore di scansione.

Code coverage non totale. Se l'approccio di *fuzzing* operato da Bouncer per analizzare dinamicamente il codice può essere giudicato accettabile da un punto di vista teorico (in un'app reale il numero di flussi di esecuzione è elevato ed è impossibile controllarli tutti; Bouncer realisticamente si limita a verificare quelli principali, statisticamente di più facile accesso), dal punto di vista pratico vi è la possibilità per un autore di malware di codificare le funzionalità malevole in uno o più flussi di esecuzione non richiamati frequentemente per evitare il rilevamento.

Facilità di rilevazione dell'emulatore. Non sono stati mascherati i numerosi dettagli (presenza del file `/sys/qemu_trace`, codename della CPU pari a `goldfish`, numero seriale/modello del dispositivo apparentemente costanti) che consentono a un'applicazione di determinare se l'esecuzione sta avvenendo in un ambiente emulato e non su un dispositivo reale. Come fatto notare da Oberheide e Miller, sarebbe inoltre possibile determinare la specifica versione di QEMU utilizzata [27] e sfruttare un *exploit* mirato per abusare l'infrastruttura di controllo stessa.

IP del dispositivo in netblock fissi. L'indirizzo di rete del dispositivo emulato appartiene a *netblock* pubblicamente registrati (dunque individuabili) a nome di Google e (finora) mai variati. Tale falla, analizzata da entrambi i lavori presentati, potrebbe essere sfruttata da malware reali.

Dati sul dispositivo emulato non realistici. L'account Google associato al dispositivo potrebbe essere utilizzato come *fingerprint* per determinare se eseguire codice maligno o meno. Un analogo ragionamento vale se si considerano numero di telefono, numero seriale del dispositivo, contatti presenti, file salvati, assenza di SMS/registro chiamate.

Bassa sensibilità dell'engine. Il motore di scansione si è rivelato poco sensibile all'esecuzione di codice malevolo o quantomeno sospetto, soprattutto se eseguito a periodicità elevate.

Un fattore proprio della piattaforma ha inoltre giocato un ruolo cruciale:

Assenza di contesto e concessione "tutto o nulla". Android richiede all'utente di concedere i permessi a un'applicazione solo al momento della sua installazione sul dispositivo, e impone che tali permessi vengano richiesti tutti in una sola occasione [16]. Questo impedisce all'utente di discriminare i permessi necessari da quelli legati a una singola funzionalità, magari facoltativa, dell'app, e di conseguenza di concedere esclusivamente i permessi che l'utente reputa essenziali[7]. Ciò si differenzia dagli approcci adottati da altri sistemi operativi mobile, in particolar modo iOS, per cui permessi particolarmente sensibili (quali la geolocalizzazione) vengono concessi dall'utente a tempo di esecuzione [18].

Per correggere i punti deboli appena esposti la ricerca si è mossa nelle seguenti direzioni.

Migliorare la gestione dei permessi. È possibile migliorare la gestione dei permessi attualmente esistente discriminando tra quelli essenziali e quelli richiesti solo per alcune funzionalità aggiuntive; un utente che non abbia bisogno di queste ultime, o che ne faccia un uso sporadico, potrebbe disattivarle. Un altro punto debole dell'implementazione corrente è la bassa specificità dei permessi: associando questi ultimi a componenti o flussi di informazione specifici è possibile discriminare più efficacemente il malware.

Potenziare l'analisi statica. Si possono migliorare gli algoritmi di ricerca dei pattern e degli exploit più comuni e, auspicabilmente, condividere all'interno dell'industria e con i ricercatori le relative definizioni (un primo passo in questo senso è stato compiuto dall'Android Malware Genome Project [33, 34]). Ciò sarebbe utile per implementare un primo livello di protezione (di minor impatto sulle risorse rispetto all'analisi dinamica).

Potenziare l'analisi dinamica. Grazie alla separazione in componenti delle funzionalità delle applicazioni, nonché al fatto che la maggior parte delle volte è necessario sfruttare le API del sistema operativo per mettere in atto comportamenti dannosi, è possibile migliorare l'analisi dinamica delle app.

5 Security Enhanced Android Framework

Il *Security Enhanced Android Framework* [2] è un'estensione del modello di controllo degli accessi che si propone di modificare la gestione attuale per tenere conto non più dei singoli permessi richiesti da un'applicazione, ma delle loro *sequenze*. In questo modo si aggiungono informazioni contestuali al modello tradizionale.

La prima differenza sostanziale è la possibilità di abilitare (concedere) solo un sottoinsieme P' dei permessi

[7] Fonti non ufficiali riferiscono che nell'ultima versione di Android (4.3) è presente, seppur nascosta, una funzionalità per l'attivazione e la disattivazione dei singoli permessi richiesti da ogni app [1].

P richiesti dall'applicazione nel relativo manifesto. Ciò risolve il problema della concessione "tutto o nulla".

Per la valutazione delle sequenze di richieste dei permessi il framework crea un *grafo* (diretto aciclico) *del comportamento dell'applicazione a tempo di esecuzione* (*application runtime behavior graph*) in cui:

- ogni nodo corrisponde a uno stato dell'applicazione considerata[8];
- ogni arco rappresenta il passaggio da uno stato a un altro, operato richiedendo un particolare permesso (indicato nell'etichetta dell'arco medesimo).

Le sequenze di richieste di permessi considerate sono dunque rappresentate formalmente da cammini sul grafo in cui, ai fini della valutazione, si ignorano i nodi di partenza e di arrivo; in altri termini, se l'insieme

$$P^i = \begin{cases} \{p\colon p \text{ è un permesso}\} & \text{se } i = 1 \\ \{p_1 \to p_2\colon p_1 \in P^{i-1}, p_2 \in P^1\} & \text{se } i > 1 \end{cases}$$

definisce le sequenze di lunghezza i, ogni sequenza considerata appartiene a $P^+ = \bigcup_{i=1}^{+\infty} P^i$. Si definiscono poi le sequenze potenzialmente malevole (in grado di causare violazioni di confidenzialità) tramite una funzione $\delta\colon P^+ \mapsto \{H, B, H \vee B\}$, dove H *(harmful)* denota una sequenza sicuramente malevole, B *(benign)* una sicuramente non malevola, $H \vee B$ una sequenza dubbia per cui l'utente è chiamato a decidere in merito. L'approccio, pur essendo generale, viene adottato solo per sequenze binarie o ternarie.

Si considerano poi due repository:

permission repository, contenente i singoli permessi richiesti da ogni applicazione (estratti dal relativo manifesto), nonché le sequenze di permessi richieste da essa e le decisioni prese dall'utente solo nel corso della sessione corrente (tempo di vita dell'attività principale); esso contiene anche le specifiche dei sottoinsiemi di permessi concessi dall'utente e un *flag* che specifica se abilitare il monitoraggio per tale applicazione (in questo modo è possibile disabilitare i controlli per le applicazioni di sistema, limitando l'impatto sulle performance);

policy repository, contenente i *pattern* (sequenze) riconosciuti sicuramente dannosi. Alcuni esempi in merito sono i seguenti:

`RECORD_AUDIO` → `INTERNET`

`READ_CONTACTS` → `INTERNET` → `SEND_SMS`

`ACCESS_FINE_LOCATION` → `SEND_SMS`

`RECEIVE_SMS` → `WRITE_SMS` → `SEND_SMS`

`READ_CONTACTS` → `INTERNET` → `CALL_PHONE`

Il *reference monitor* è costituito da un componente denominato *policy evaluator*; esso si occupa, note le sequenze rivelatesi necessarie a tempo di esecuzione, di generare tutte le sottosequenze binarie o ternarie possibili (mantenendo l'ordine cronologico dei singoli permessi) per poi operare il confronto con i repository e visualizzare una notifica all'utente in caso di riscontro positivo (questi ha la facoltà di autorizzare o meno la prosecuzione dell'esecuzione).

Pur rappresentando un notevole miglioramento rispetto alla gestione dei permessi così come implementata nelle attuali versioni di Android, l'approccio presenta ancora dei limiti evidenti.

Possibile bassa accettabilità psicologica. Alcune delle sequenze riconosciute come dannose sono in realtà comuni per determinate tipologie di applicazioni: sia sufficiente considerare la sequenza di richieste `RECORD_AUDIO` → `INTERNET` propria delle applicazioni VoIP. Il numero elevato di avvisi in tali casi potrebbe indurre l'utente a ignorarli.

Necessità di aggiornare il policy repository. Il policy repository deve essere aggiornato frequentemente per includere nuove sequenze di permessi riconosciute come dannose.

Assenza di controlli sull'integrità. Le sequenze malevole attualmente riconosciute tali si riferiscono a violazioni di confidenzialità; non vi sono invece protezioni contro attacchi mirati a violare l'integrità del sistema (ad esempio la sostituzione di un'app legittima con una malevola).

App di sistema non protette. L'aver consentito la possibilità di disabilitare la protezione per le applicazioni di sistema (per ragioni prestazionali) consente a queste ultime di richiedere qualunque sequenza di permessi (concessa dal manifesto) e di ottenerli, senza che il framework sia in grado di impedirlo. Ciò può essere un veicolo di attacco, come dimostrato dalla recente scoperta di Jeff Forristal (una vulnerabilità nella validazione delle firme digitali dei package `.apk` che può condurre all'acquisizione di privilegi di root sfruttando app di sistema) [8].

Non resistenza a richieste separate dei permessi. L'aver limitato l'analisi delle sequenze dei permessi ai casi binario e ternario consente di portare a termine attacchi inserendo fra i permessi dannosi altre richieste che non lo sono; si può sfruttare una debolezza analoga eseguendo la prima azione dannosa in una sessione, salvando i dati, attendendo l'inizio di una sessione successiva (è sufficiente terminare un'attività ed eseguirne un'altra, ad esempio sostituendo la schermata attiva), recuperando le informazioni e portando a termine la seconda azione. Non è chiaramente possibile aumentare la dimensione delle richieste analizzate in quanto l'analisi, dovendo generare e valutare tutte le sottosequenze, richiederebbe tempo esponenziale[9]; una possibile soluzione sarebbe generare le sottosequenze rimuovendo le combinazioni di permessi sicuramente non dannose.

[8] Nell'articolo che introduce il framework non è definito formalmente il concetto di stato; lo possiamo però immaginare come una rappresentazione del contenuto dei registri della CPU e della memoria.

[9] In [2] è disponibile un'analisi temporale dettagliata da cui risulta che la generazione è il passo con maggiori richieste computazionali.

6 CopperDroid

CopperDroid è una piattaforma per l'analisi dinamica di malware per Android [31]. Essa si pone l'obbiettivo di filtrare le applicazioni indipendentemente dal linguaggio in cui sono scritte (Java — linguaggio d'elezione — o codice nativo C/C++) ed indipendentemente dalla piattaforma su cui deveono essere eseguite (virtual machine Dalvik o esecuzione di binari ELF). In particolare essa si basa su tre principi.

- Analisi a basso livello (specifica per l'OS) e analisi ad alto livello (specifica per la virtual machine Dalvik).
- Stimolazione delle applicazioni per raggiungere tutti gli *entry point* e per garantire la più ampia *code coverage* possibile.
- Testing su database di malware noti e scansioni via Web di applicazioni segnalate dagli utenti.

L'architettura del sistema si fonda su un emulatore Android basato su QEMU e appositamente modificato al fine di poter analizzare a basso livello il codice effettivamente eseguito.

6.1 Architettura

CopperDroid sfrutta il protocollo RSP (remote serial protocol) del debugger GNU GDB per consentire una connessione diretta tra host e ambiente emulato (virtual machine introspection). Il pregio di questo meccanismo consiste nell'assoluta invisibilità alle applicazioni in esecuzione nel guest e la non modifica del flusso di esecuzione. CopperDroid controlla se e quando viene effettuata una chiamata di sistema filtrando queste ultime in base al grado di importanza rispetto alla privacy dell'utente (sono infatti implementate due specifiche categorie per le chiamate sensibili: *to-be-tracked* e *to-be-monitored*)[10].

6.2 Funzionamento

Il sistema Android e le applicazioni si basano fortemente sulla comunicazione inter-processo (IPC) e sulle chiamate remote di procedure (RPC). CopperDroid è il primo sistema che cerca di effettuare un'analisi dettagliata di questi canali di comunicazione. Ad alto livello, ad esempio, inviare un SMS corrisponde ad ottenere un'istanza di `smsManager` ed invocarne il metodo `sendTextMessage`; questo si riflette sull'invocazione della funzione `sendText` del servizio di Binder denominato `isms`. La stessa operazione vista a basso livello consiste invece nell'effettuare due chiamate `ioctl` di sistema su `/dev/binder`: la prima per identificare il servizio, la seconda per invocare il metodo. Il software introspeziona ogni chiamata `ioctl` per ricostruire l'invocazione remota consentendo di stabilire di fatto quali siano i parametri utilizzati.

[10] L'articolo in cui viene presentata questa piattaforma [31] non precisa quali chiamate appartengano a questi due gruppi; si suppone che ciò sia stato fatto per rendere più difficile l'aggiramento del motore di scansione da parte degli autori di malware.

6.2.1 Copertura degli entry point

Non sempre un'applicazione viene eseguita invocando il suo metodo `main`: in Android un'app può avere più di un entry point (in genere ve ne è uno per ogni attività possibile, ad esempio uno per l'avvio normale dell'applicazione, un secondo per l'avvio da un particolare tipo di notifica, e così via), chiamati dall'applicazione o dal sistema stesso. CopperDroid cerca di mappare questi punti di ingresso partendo dal manifesto dell'applicazione, in particolare esaminando i permessi richiesti e generando un elenco di *entry point* possibili. Vengono quindi creati degli eventi fasulli che stimolano il gestore degli eventi dell'applicazione causando un'esecuzione secondaria della stessa.

6.3 Risultati

La valutazione di CopperDroid è stata effettuata su due database di malware Android disponibili online: *Android Malware Genome Project* [33] e *Contagio* [28]. Sull'emulatore CopperDroid è stata montata un'immagine pulita di Android 2.3.3, inserendo al suo interno informazioni personali, SMS, immagini e popolando i registri delle chiamate. Alla fine di ogni analisi il sistema è stato ripristinato a uno stato pulito per impedire infezioni da parte dei malware.

L'emulatore è stato testato su oltre 1600 campioni dimostrando un incremento della percentuale di riconoscimento del 22% per quanto riguarda il database *Android Malware Genome Project* e del 28% per quanto riguarda il database *Contagio*.

7 Conclusioni

In questa relazione abbiamo esaminato i principali fattori di rischio di Android e analizzato nel dettaglio due possibili contromisure alla sempre maggiore diffusione del malware su tale piattaforma. Questi ultimi sistemi sono radicalmente diversi (il primo è un potenziamento dell'execution monitor attualmente esistente, il secondo è un analizzatore dinamico), così come sono diversi gli approcci che si dovrebbero adottare per la difesa dal codice malevolo (*defense in depth*): le applicazioni dovrebbero essere filtrate più incisivamente sullo store ufficiale (eliminando la facile rilevabilità dell'emulatore usato in Bouncer), quindi ulteriormente limitate (il problema principale, a giudizio degli autori, è la bassa specificità dei permessi, a cui si potrebbe porre rimedio tracciando i flussi d'informazione). Sarebbe anche auspicabile che i produttori di dispositivi prestassero maggior attenzione al rilascio tempestivo degli aggiornamenti del sistema operativo; considerata la diffusione dei malware per Android, un lieve aumento del costo finale dei dispositivi a fronte della garanzia della fornitura di immagini con nuove versioni dell'OS sarebbe considerato vantaggioso.

Riferimenti bibliografici

[1] R. Amadeo. «App Ops: Android 4.3's Hidden App Permission Manager, Control Permissions For Individual Apps!» Disponibile a `http://`

www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-app-permission-manager-control-permissions-for-individual-apps/. Lug. 2013.

[2] Hammad Banuri et al. «An Android runtime security policy enforcement framework.» In: *Personal and Ubiquitous Computing* 16.6 (2012), pp. 631–641. URL: http://dblp.uni-trier.de/db/journals/puc/puc16.html#BanuriAKMAKYTAAZ12; http://dx.doi.org/10.1007/s00779-011-0437-6; http://www.bibsonomy.org/bibtex/2f6c7d1ab8caeee8319af3bbb60fe66a5/dblp.

[3] J. Case. «Exclusive: Vulnerability In Skype For Android Is Exposing Your Name, Phone Number, Chat Logs, And A Lot More». Disponibile a http://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/. Giu. 2012.

[4] L. Cavallaro. «CopperDroid (Part 2)». In: *Malicious Software and its Underground Economy: Two Sides to Every Story*. Disponibile a https://d396qusza40orc.cloudfront.net/malsoftware/w4/w4-4_copperdroid2.pdf. Lug. 2013.

[5] «CyanogenMod». Disponibile a http://www.cyanogenmod.org/.

[6] A. Desnos e G. Gueguen. «Android: From Reversing To Decompilation». In: *Black Hat Abu Dhabi 2011*. Disponibile a https://androguard.googlecode.com/files/bh2011.pdf. Dic. 2011.

[7] W. Enck e P. McDaniel. «Understanding Android's Security Framework». Disponibile a http://siis.cse.psu.edu/slides/android-sec-tutorial.pdf. Ott. 2008.

[8] J. Forristal. «Android: One Root to Own Them All». In: *Black Hat USA 2013*. Disponibile a https://media.blackhat.com/us-13/US-13-Forristal-Android-One-Root-to-Own-Them-All-Slides.pdf. Lug. 2013.

[9] Google. «Android Developers: Android Compatibility». Disponibile a http://source.android.com/compatibility/index.html.

[10] Google. «Android Developers: Compatibility Program Overview». Disponibile a http://source.android.com/compatibility/overview.html.

[11] Google. «Android Developers: Dashboards». Disponibile a https://developer.android.com/about/dashboards/index.html.

[12] Google. «Android Developers: Ice Cream Sandwich». Disponibile a https://developer.android.com/about/versions/android-4.0-highlights.html.

[13] Google. «Android Developers: Jelly Bean». Disponibile a https://developer.android.com/about/versions/jelly-bean.html.

[14] Google. «Android Developers: Licenses». Disponibile a http://source.android.com/source/licenses.html.

[15] Google. «Android Developers: Manifest.permission». Disponibile a https://developer.android.com/reference/android/Manifest.permission.html.

[16] Google. «Android Developers: Permissions». Disponibile a https://developer.android.com/guide/topics/security/permissions.html.

[17] M. E. Gordon. «The iOS and Android Two-Horse Race: A Deeper Look into Market Share». Disponibile a http://blog.flurry.com/bid/97860/The-iOS-and-Android-Two-Horse-Race-A-Deeper-Look-into-Market-Share.

[18] Apple Inc. «iOS 6.0 Release Notes». Disponibile a https://developer.apple.com/library/ios/releasenotes/General/RN-iOSSDK-6_0/.

[19] H. Lockheimer. «Android and Security». Disponibile a http://googlemobile.blogspot.it/2012/02/android-and-security.html. Feb. 2012.

[20] Lookout. «Inside the Android Security Patch Lifecycle». Disponibile a https://blog.lookout.com/blog/2011/08/04/inside-the-android-security-patch-lifecycle/.

[21] D. Maslennikov. «Mobile Malware Evolution: An Overview, Part 4». Disponibile a https://www.securelist.com/en/analysis/204792168/Mobile_Malware_Evolution_An_Overview_Part_4. Mar. 2011.

[22] D. Maslennikov. «Mobile Malware Evolution, Part 5». Disponibile a https://www.securelist.com/en/analysis/204792222/Mobile_Malware_Evolution_Part_5. Feb. 2012.

[23] D. Maslennikov. «Mobile Malware Evolution, Part 6». Disponibile a https://www.securelist.com/en/analysis/204792283/Mobile_Malware_Evolution_Part_6. Feb. 2013.

[24] J. Oberheide e C. Miller. «Dissecting the Android Bouncer». In: *SummerCon 2012*. Disponibile a http://jon.oberheide.org/files/summercon12-bouncer.pdf. Brooklyn, NY, giu. 2012.

[25] OpenSignal. «Android Fragmentation Visualized». Disponibile a http://opensignal.com/reports/fragmentation-2013/. Lug. 2013.

[26] OpenSignal. «Android Fragmentation Visualized: The many faces of a little green robot». Disponibile a http://opensignal.com/reports/fragmentation.php. 2012.

[27] R. Paleari et al. «A fistful of red-pills: How to automatically generate procedures to detect CPU emulators». In: *WOOT '09 – 3rd USENIX Workshop on Offensive Technologies*. Disponibile a http://static.usenix.org/event/woot09/tech/full_papers/paleari.pdf. Ago. 2009.

[28] M. Parkour. «Contagio Mobile». http://contagiominidump.blogspot.com.

[29] N. J. Percoco e S. Schulte. «Adventures in Bouncerland: Failures of Automated Malware Detection within Mobile Application Markets». In: *Black Hat USA 2012*. 2012.

[30] «Pimp My ROM». Disponibile a `http://pimpmyrom.org/`.

[31] Alessandro Reina, Aristide Fattori e Lorenzo Cavallaro. «A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors». In: *Proceedings of the 6th European Workshop on System Security (EUROSEC)*. Prague, Czech Republic, apr. 2013.

[32] The Citizen Lab Targeted Threats team. *Permission to Spy: An Analysis of Android Malware Targeting Tibetans*. Rapp. tecn. 16. University of Toronto, apr. 2013.

[33] Y. Zhou e X. Jiang. «Android Malware Genome Project». Disponibile a `http://www.malgenomeproject.org/`.

[34] Yajin Zhou e Xuxian Jiang. «Dissecting Android Malware: Characterization and Evolution.» In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012, pp. 95–109. ISBN: 978-0-7695-4681-0. URL: `http://dblp.uni-trier.de/db/conf/sp/sp2012.html#ZhouJ12;http://doi.ieeecomputersociety.org/10.1109/SP.2012.16;http://www.bibsonomy.org/bibtex/21633cd31ac0e2d92b13e1ec7573cfff7/dblp`.