Pattern Recognition
and Applications Lab

Sicurezza dei Sistemi Android

Dott. Ing. Davide Maiorca, Ph.D.

davide.maiorca@diee.unica.it

Corso di Sicurezza Informatica – A.A. 2015/2016

Università
di Cagliari, Italia

Dipartimento di
Ingegneria Elettrica
ed Elettronica

Sommario

- **Introduzione**
- **Basi di Android**
- **Sicurezza**
  - Permessi
  - Malware
  - Offuscamento
- **Ricerca sulla Sicurezza**
  - Sicurezza Antivirus
  - Attacchi a Sistemi di Analisi
- **Conclusioni**

Introduzione

Introduzione

- L'ultima parte del corso vuole fornire una panoramica sulla sicurezza del **mondo mobile**
- In particolare, ci concentreremo sul sistema operativo Android
- Android è, senza dubbio, il sistema operativo mobile più utilizzato…
- …Ed il fatto che il sistema sia totalmente open source comporta maggiori **possibilità di attacco** rispetto ad IOS
- Ripasseremo, prima di tutto, **le basi del sistema operativo** Android
- Poi vedremo le caratteristiche principali dei malware ed alcuni dei contributi forniti dalla ricerca scientifica nell'analisi di questi
- Infine, forniremo una panoramica del lavoro di ricerca svolto dal PraLAB su Android
  - **Possibilità di tesi di laurea!**

Introduzione

1 Miliardo di dispositivi

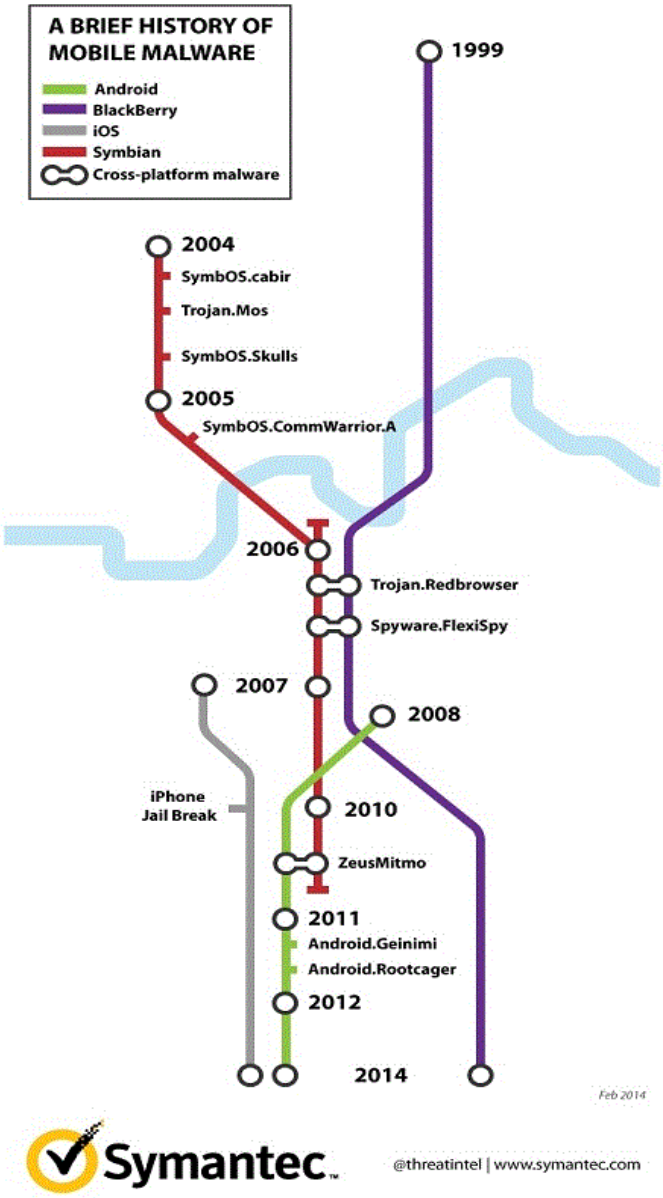
1.500.000 Applicazioni

Più di 5.000.000 di malware!

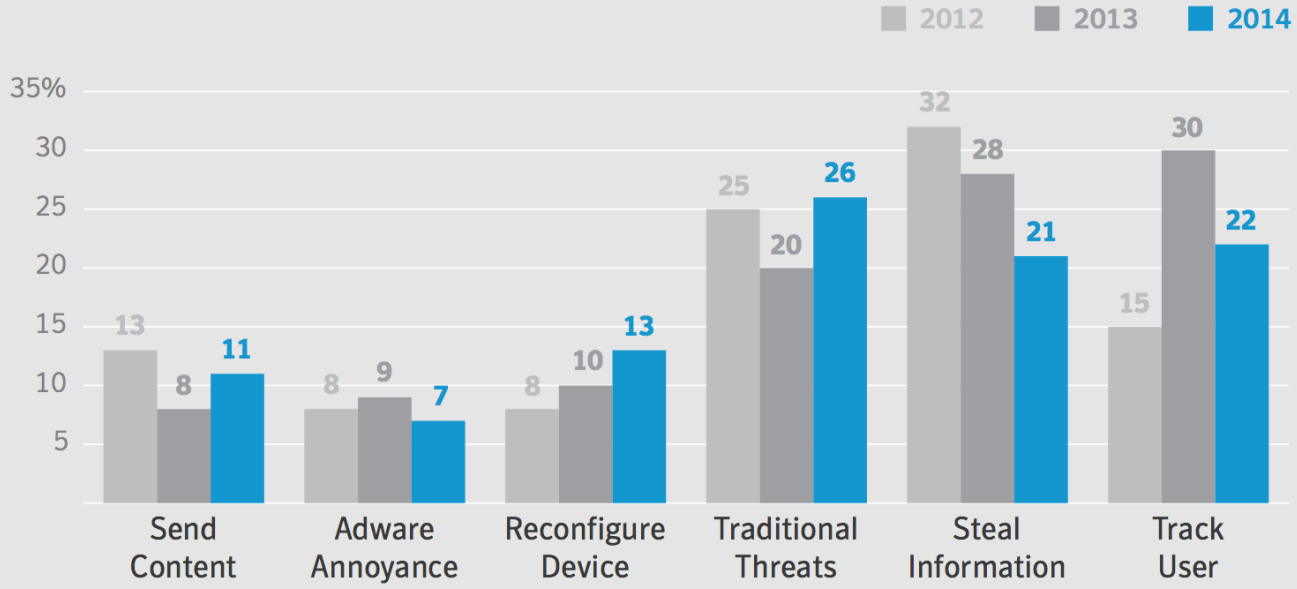
Introduzione – Sicurezza Android

- Molte applicazioni non vengono scaricare dal market standard di Android (Google Play), ma anche da market di terze parti
  - Prezzi più vantaggiosi
  - Applicazioni che non si trovano sullo store ufficiale
- Questo si combina col fatto che, molto spesso, l'operazione svolta più spesso per violare la sicurezza di Android è quella di prendere i permessi di root
  - Eseguita dallo stesso utente!
  - Consente di prendere il controllo completo delle funzionalità del telefono
  - Lo vedremo nel dettaglio più avanti nella lezione
- Questi due elementi hanno portato al diffondersi di malware attraverso, principalmente, stores di terze parti
  - Molti sfruttano il fatto che la vittima abbia permessi di root
  - Altri, anche se tali permessi non sono presenti, creano numerosi danni, soprattutto economici

Introduzione – Mobile Malware



Introduzione – Mobile Malware



Send Content – Threats that send premium SMS, Spam and SEO Poisoning threats.

Adware/Annoyance – Threats that cause advertisement popups and unwanted information.

Reconfigure Device – Threats that modify user settings, and elevates privileges.

Traditional Threats – Threats like Backdoor Trojans, Downloaders, DDoS utility, Hacktool and Security Alerts.

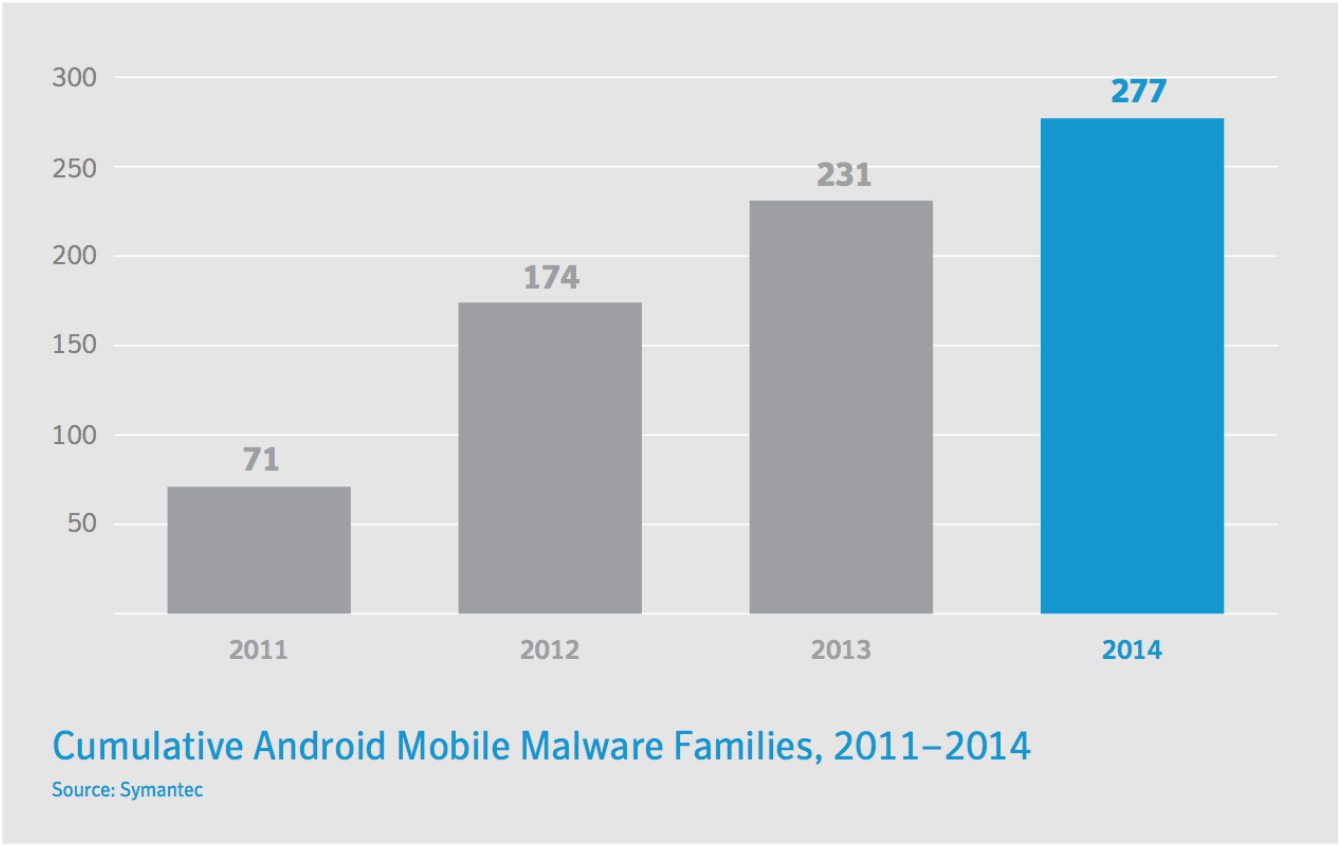
Steal Information – Threats that steal device data, media files and any user credentials. Eg., Banking Trojan.

Track User – Threats that spy on users, tracks user location.

Mobile Threat Classifications, 2012–2014

Source: Symantec

Introduzione – Mobile Malware



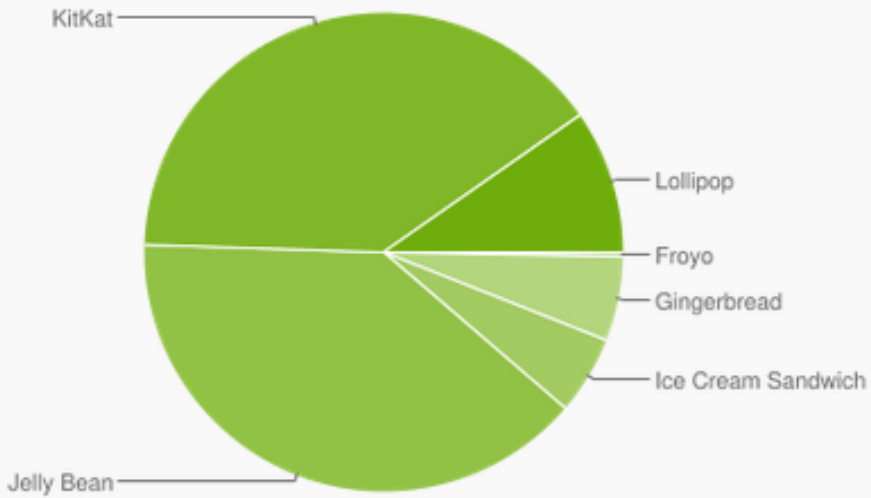
Cumulative Android Mobile Malware Families, 2011–2014

Source: Symantec

Elementi di Android

Versioni Android

Version	Codename	API	Distribution
2.2	Froyo	8	0.3%
2.3.3 - 2.3.7	Gingerbread	10	5.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	5.3%
4.1.x	Jelly Bean	16	15.6%
4.2.x		17	18.1%
4.3		18	5.5%
4.4	KitKat	19	39.8%
5.0	Lollipop	21	9.0%
5.1		22	0.7%



Struttura Generale

- Il sistema operativo Android si basa su una architettura a strati
- Sono presenti 4 strati fondamentali (impilati a “stack”)
- Funzioni del telefono (rubrica, contatti, etc.)
- Application Framework (gestione delle attività, risorse, content providers, etc.)
- Librerie native e codice runtime (Dalvik Virtual Machine / ART)
- Kernel di Linux (funzioni fondamentali del sistema)

Architettura a strati

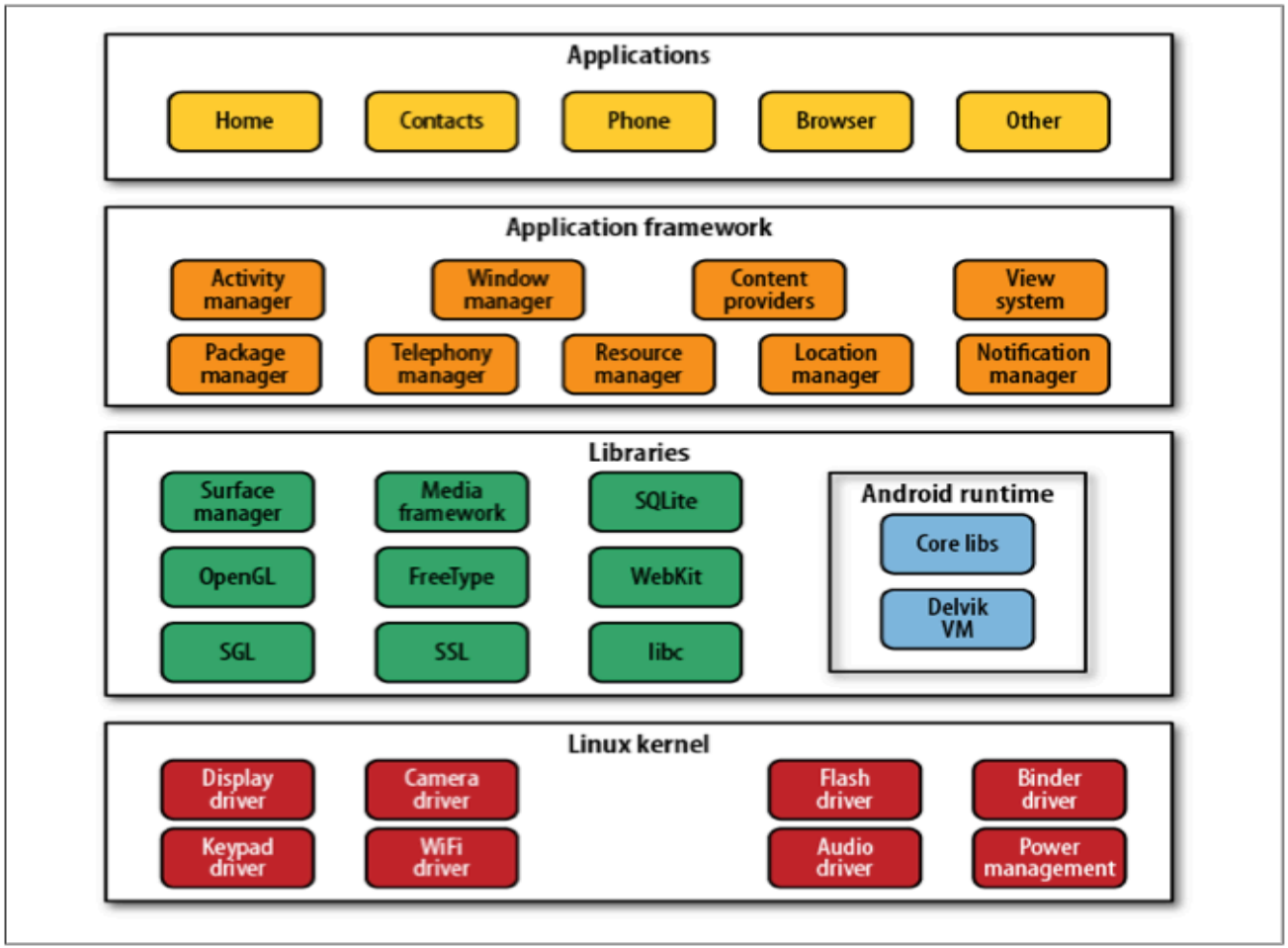


Figure 2-1. Android stack

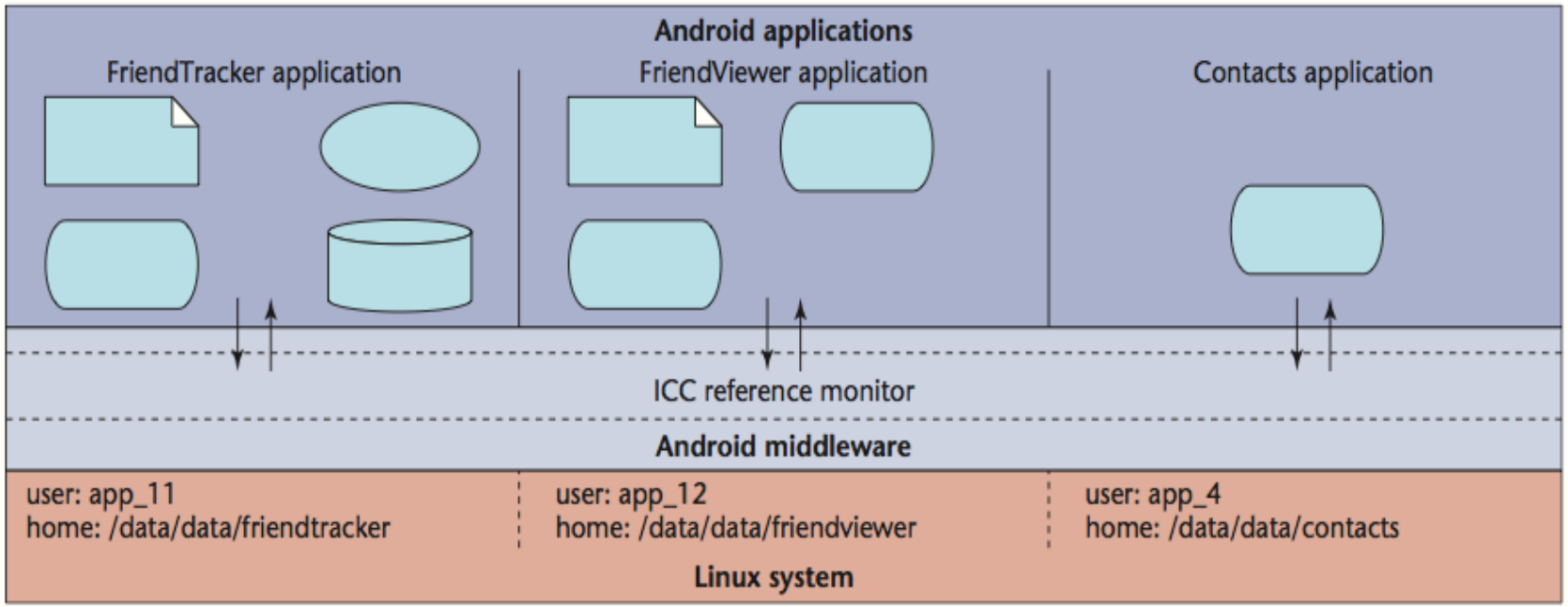
Strato 1 - Kernel

- Gestisce I processi
- Fornisce I Driver per l'accesso a risorse fisiche
  - L'uso dei driver è abilitato da chiamate di sistema
- Supporta comunicazione fra processi (IPC)
  - Driver Binder (Intents)
  - Sockets
  - Binder (a livello di kernel)
  - Intents (ad esempio: dichiarare l'intenzione di usare un browser esterno)
  - Interazione con i Content Providers per la gestione dei dati
- Possibilità di firmare digitalmente le app mediante *certificati*
- File system criptato (Kernel >=3.0, AES 128 bit)
- Protezione con password per accesso al dispositivo
- Protezione contro errori di memoria!
  - DEP (Data Execution Prevention)
  - Da Ice Cream Sandwich è stato introdotto ASLR

Sandboxing

- Ogni applicazione viene eseguita in un ambiente “isolato”
- Ogni app accede solo al suo insieme di risorse
  - Una app NON può modificare i dati su cui agisce un’altra app
  - Se una app viene compromessa, le altre app non ne vengono influenzate
- Ogni applicazione ha il suo Linux User ID (FSUID)
  - Impedisce che le conseguenze di un exploit contro una app possano “propagarsi” ad altre app
- Ma è sicura?
- Se due applicazioni condividessero lo stesso FSUID potrebbero condividere le risorse
  - E’ possibile definire una condivisione di user id! (shareduserid)

Sandboxing


Android applications
FriendTracker application
FriendViewer application
Contacts application
ICC reference monitor
Android middleware
user: app_11
home: /data/data/friendtracker
user: app_12
home: /data/data/friendviewer
user: app_4
home: /data/data/contacts
Linux system

Strato 2 – Librerie Native & Runtime

- Scritte in C/C++ (da **NON confondere** con quelle usate in Java per programmare il codice…)
- Si interfacciano con l'application framework
  - **Libc**
  - **SQLLite**
  - **OpenGL**
  - **E molte altre!**
- Sono l'unico strumento con cui un attaccante può direttamente accedere alla memoria durante l'esecuzione di un programma…
- Per eseguire le applicazioni, Android ricorre ad una specifica runtime
- Fino alla versione 4.2 si utilizza la **Dalvik Virtual Machine**
- Dalla versione 4.4 (e obbligatoriamente nella 5) si utilizza **ART (Android RunTime)**

Dalvik Virtual Machine

- Su Android si programma usando il linguaggio di programmazione Java…
- Ma la **Java Virtual Machine (JVM)** è poco efficiente!
- Provate a eseguirla su uno smartphone con 256 MB di memoria interna...
- Dalvik è una virtual machine simile a JVM ma molto più ottimizzata per mobile
- Compilazione **Just in Time** (istruzioni “compilate” in fase di esecuzione)
- Il codice Dalvik è ottenuto attraverso la conversione dal bytecode Java dopo la compilazione
- Efficienza molto maggiore (ottimizzato per ARM)!

Dalvik Virtual Machine (2)

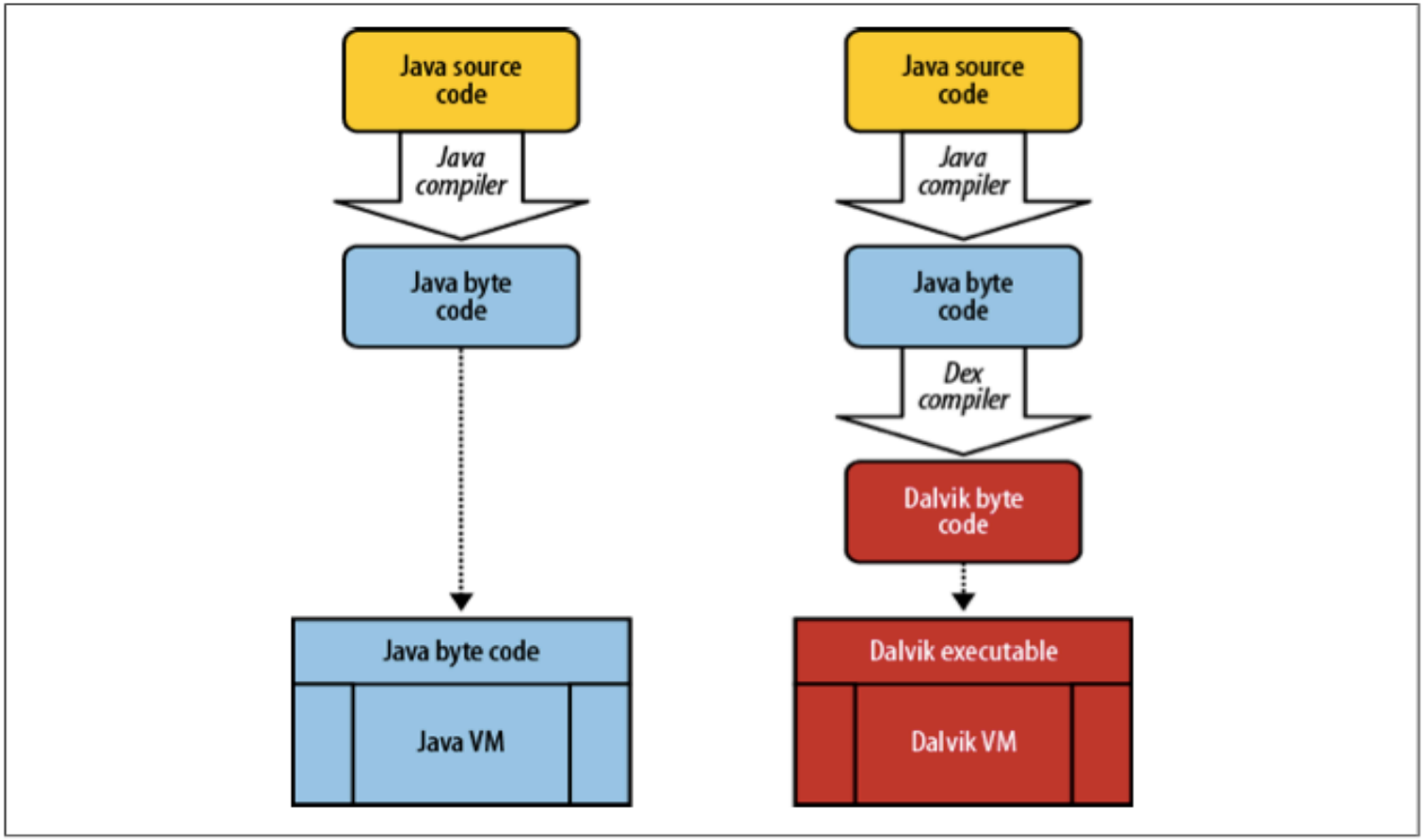
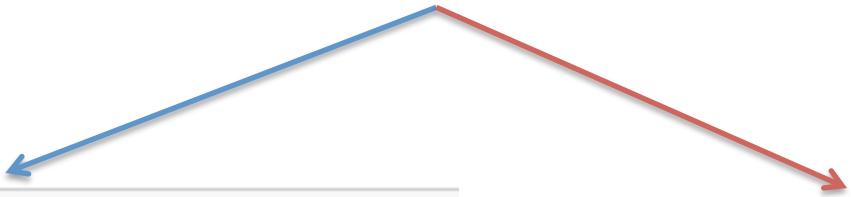


Figure 2-2. Java versus Dalvik

Dalvik Bytecode

```
public static int addConst(int val) {
   return val + 123456;
}
```



```
public static int addConst(int);
  [max_stack=2, max_locals=1, args_size=1]
   0: iload_0
   1: ldc #int 123456
   3: iadd
   4: ireturn
```

Java VM

Macchina a Stack

```
public static int addConst(int);
  [regs=2, ins=1, outs=0]
   0: const v0, #0x1E240
   1: add-int/2addr v0, v1
   2: return v0
```

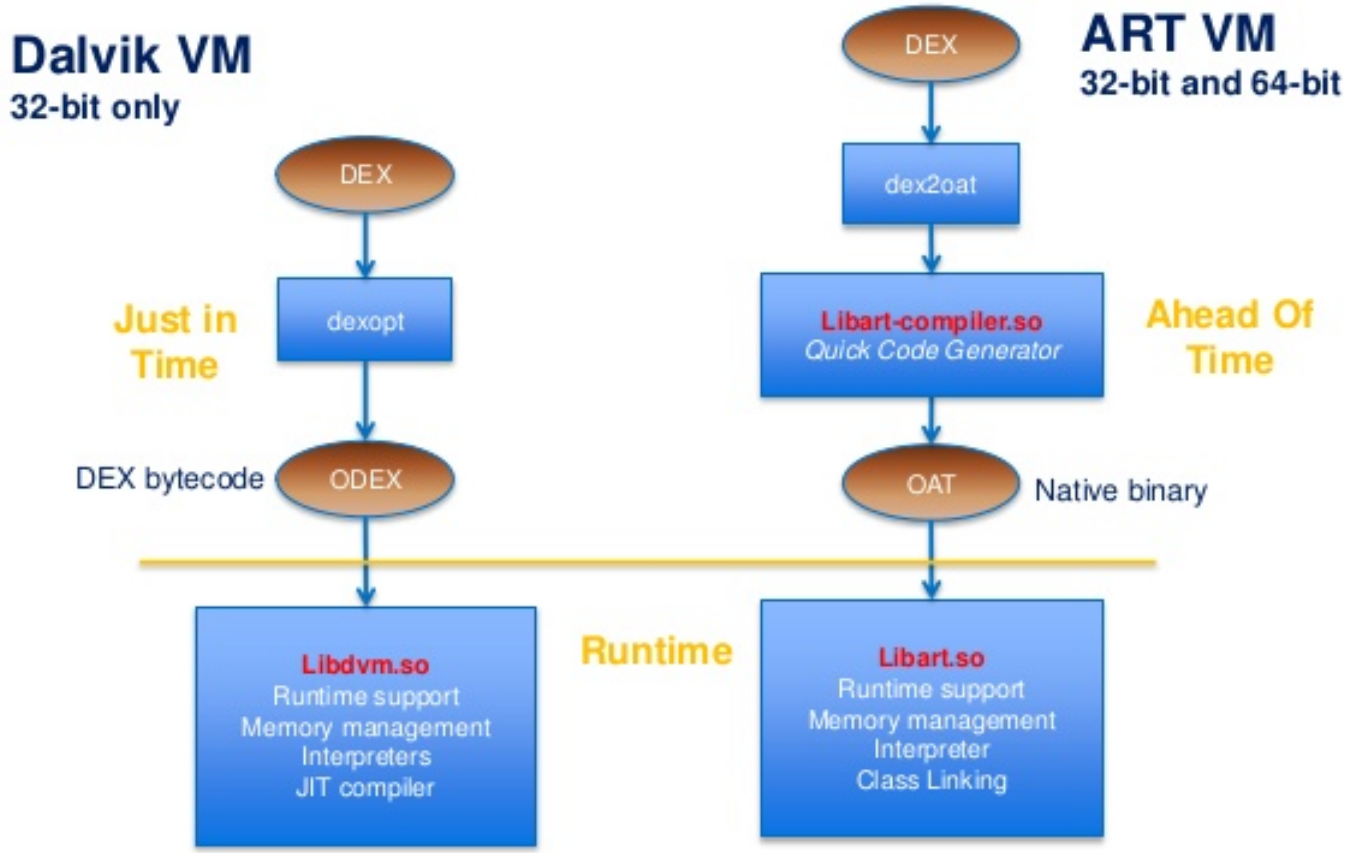
Dalvik

Macchina a registri

Android RunTime (ART)

- Evoluzione della Dalvik Virtual Machine
- Il Dalvik bytecode viene compilato e trasformato in codice macchina (ARM)
- Perciò, l'applicazione esegue direttamente delle istruzioni macchina una volta installata
  - L'architettura del processore ARM è diversa da quella intel…
- Supporta processori a 64 bit!
- Questo comporta un significativo incremento di velocità rispetto alla Dalvik Virtual Machine
- Ovviamente, il tempo di installazione è maggiore…
- E anche lo spazio occupato dall'applicazione!

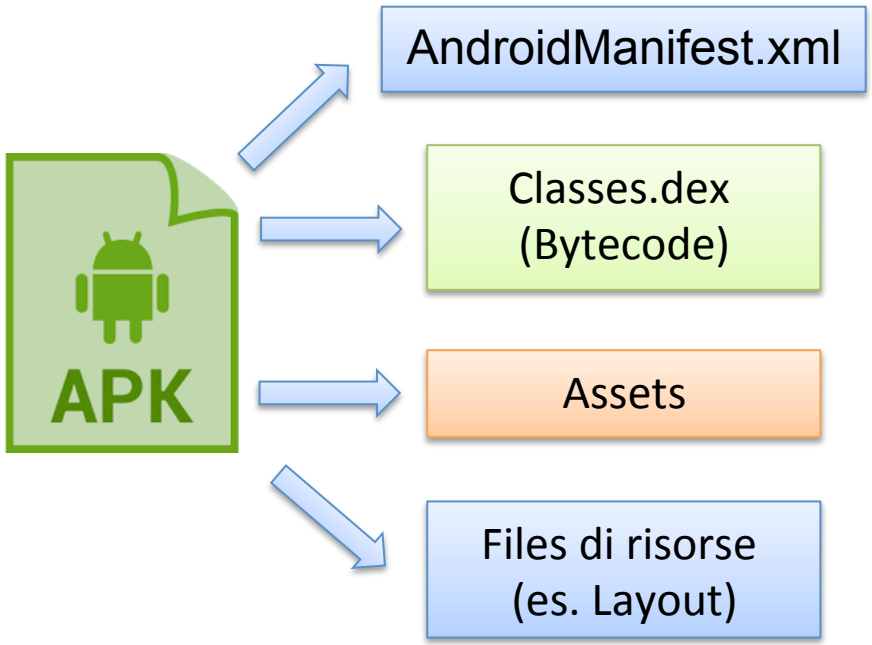
Android RunTime (ART)



Strato 3 – Application Framework

- **L'applicazione esegue le sue funzioni utilizzando quattro componenti fondamentali**
- ***Attività***
  - **Singola schermata del vostro programma**
  - **Una attività per schermata**
  - **Una applicazione ha attività multiple**
- ***Servizi***
  - **Esecuzione in background**
  - **Non hanno componenti di interfaccia**
  - **Esempio: musica mentre state usando il vostro telefono (background)**
- ***Broadcast Receivers***
  - **Il receiver è un meccanismo che resta "dormiente" e si attiva solo quando accade un certo evento del sistema a questo è "sottoscritto"**
- **Content Providers**
  - **Interfacce per la condivisione di dati fra le varie applicazioni**
  - **L'esecuzione in sandbox, infatti, fa sì che ogni applicazione lavori solo su una specifica parte delle risorse**

Strato 4 – Applicazione



Manifest.xml: File che contiene informazioni sui componenti dell'applicazione (attività, servizi…)

Classes.dex: Eseguibile Dalvik vero e proprio

Assets: Risorse esterne come immagini o altri eseguibili

Files di risorse: xml che descrivono, ad esempio, il layout dell'applicazione

Accedere al Manifest

- Data una apk, non potete visualizzare il Manifest.xml estraendolo con un normale decompressore (ad esempio unzip)
- Dovete usare un tool chiamato *apktool*
  - *Scaricabile gratuitamente: http://ibotpeaches.github.io/Apktool/*
- *java –jar apktool decode <applicazione> <output folder>*
- Oltre ad estrarre il Manifest, disassemblerà il classes.dex in un formato più "semplice" da leggere chiamato *smali*
- Una volta aperto il **Manifest**, potete visualizzare una serie di informazioni
  - Ad esempio, tutte le attività inizieranno con la parola *activity*
  - Gli *<intent-filter>* definiscono le azioni svolte dall'attività (ad esempio, se è presente main, questa sarà la PRIMA attività che verrà aperta)
  - Ricordatevi che su android NON esiste la funzione main

Accedere al classes.dex

- Per leggere il classes.dex dovete usare un tool proprietario dell'sdk di Android chiamato ***dexdump***
- ***exdump –d classes.dex***
- ***Vi fornisce la lista delle classi, dei metodi e del loro relativo bytecode***
- Il formato è un po' complesso da leggere (ad esempio, non è sempre facile capire a quali registri è associato un parametro di un metodo)
- Perciò, molto spesso, si tende a leggere il formato smali ottenuto da ***apktool***
- Più semplice capire anche la struttura del programma!
- Ma non è sempre possibile disassmeblare una applicazione…

Sicurezza

Permessi

- Come può una applicazione Android sfruttare le risorse del sistema?
- Deve chiedere il **permesso**…
- Il modello di permessi garantisce la sicurezza a livello dell'applicazione
- I permessi vengono approvati/non approvati **QUANDO VIENE INSTALLATA L'APPLICAZIONE**
- Non si può tornare indietro una volta accettati…
- Le richieste vengono definite nel **Manifest.xml** dallo sviluppatore
- Quattro diverse categorie
  - **Normal: permessi che non costituiscono un pericolo per la sicurezza**
  - **Dangerous: permessi che accedono a dati sensibili dell'utente (di questi permessi viene chiesta conferma all'utente al momento dell'installazione**
  - **Signature: permessi che possono essere ottenuti solo se l'applicazione ha lo stesso certificato del produttore del dispositivo**
  - **SignatureOrSystem: permessi usati solo nelle applicazioni presenti nell'immagine del sistema android (se, ad esempio, ci sono applicazioni prodotte da diversi produttori che hanno bisogno di condividere le stesse risorse)**

Permessi

Permission	% of apps using it
INTERNET	97.8%
READ_PHONE_STATE	93.6%
ACCESS_NETWORK_STATE	81.2%
WRITE_EXTERNAL_STORAGE	67.2%
ACCESS_WIFI_STATE	63.8%
READ_SMS	62.7%
RECEIVE_BOOT_COMPLETED	54.6%
WRITE_SMS	52.2%
SEND_SMS	43.9%
VIBRATE	38.3%
ACCESS_COARSE_LOCATION	38.1%
READ_CONTACTS	36.3%
ACCESS_FINE_LOCATION	34.3%
WAKE_LOCK	33.7%
CALL_PHONE	33.7%
CHANGE_WIFI_STATE	31.6%
WRITE_CONTACTS	29.7%
WRITE_APN_SETTINGS	27.7%
RESTART_PACKAGES	26.4%

Table 7: Top-20 most frequent permissions requested by malware (from Zhou and Jiang [18]).

Un'applicazione malevola potrebbe richiedere l'accesso a dei permessi pericolosi…

Sicuramente, se una applicazione non richiede Permessi, non potrà eseguire nessuna azione (quindi è, di fatto, innocua)

Il problema è che, molto spesso, anche le applicazioni legittime richiedono permessi pericolosi (a volte inutili)

I permessi, da soli, non bastano per capire Se una applicazione è malevola o meno

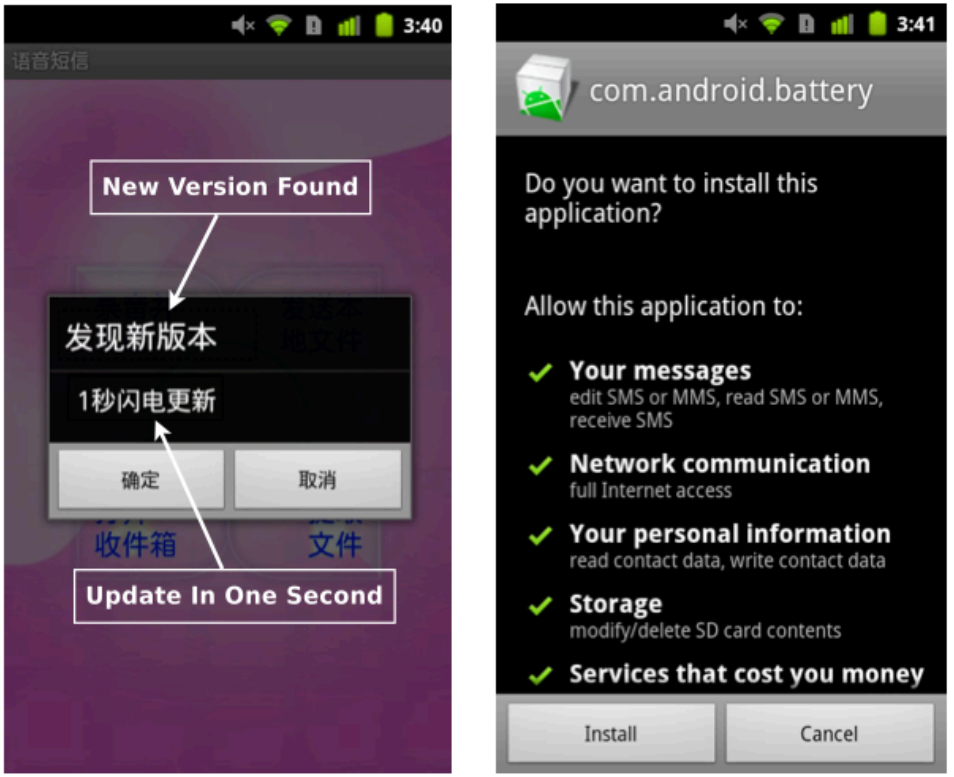
Malware

- Applicazioni che eseguono azioni malevole
  - Sms a numeri premium
  - Furto dati personali
  - Telefonate “indesiderate”
  - Trasformazione del terminale in botnet
- Più di 50 famiglie di malware diverse!
- Tecniche di creazione principali
  - Repackaging
  - Update
  - Drive-By-Download
  - Altre (stand-alone)
- I malware android possono essere raggruppati in vari modi
  - Per tecniche di attivazione (BroadCast receivers)
  - Per tipo di attacco (payload)

Tecniche di Creazione - Repackaging

- Viene iniettato un payload maligno all'interno dell'applicazione decompressa che viene poi ricompressa e firmata
- Infine, viene sottomessa ad uno store (solitamente di terze parti)
- Possono essere sfruttati dei permessi contenuti nell'applicazione principale (permessi dangerous)
- E' la tecnica più usata per la sua semplicità e per la disponibilità di frameworks per l'attacco
  - AFE (Android Framework For Exploitation)
- Numerosi lavori per individuare il repackaging sono stati effettuati a livello di ricerca scientifica

Tecniche di Creazione - Update



(a) The Update Dialogue (b) Installation of A New Version

Figure 2. An Update Attack from `BaseBridge`

Dopo aver installato l'applicazione (legittima), vi viene chiesto di scaricare un aggiornamento

L'aggiornamento contatta un url malevolo che scarica una app malicious che viene poi eseguita

N.B. a volte l'app "aggiornata" può essere contenuta dentro l'applicazione stessa!

Questa tecnica è utilizzata da alcuni tra i malware mobile più famosi (es. Base Bridge e Droid Kung Fu)

Tecniche di Creazione – Drive-By Download e altre

- Stessa tecnica usata nei malware comuni
- Un advertisement vi reindirizza verso un url maligno
- Dall'url verrà scaricata una app maligna che verrà eseguita
- Esempi malware:
  - GGTracker
  - JIFake
- Oltre alle tecniche viste finora, un malware può essere progettato "stand alone"
  - Spyware
  - Malware che usano interfacce simili ad applicazioni legittime (nota: NON SONO REPACKAGED! Esempio: FakeNetflix)
  - Applicazioni con funzionalità ibride (eseguono funzionalità legittime reali, miste a malicious)

Tecniche di Attivazione e Tipi di Attacco (1)

- Alcuni malware, per attivarsi, sfruttano il meccanismo dei "broadcast receivers e si attivano ricevendo dei segnali
  - BOOT_COMPLETED (Avvio del sistema)
  - SMS_RECEIVED (Rimuove messaggi verso numeri premium in modo che l'utente non si accorga di essere infetto)
  - …e diversi altri!
- Vediamo, ora, le principali tipologie di attacco
- *Root Exploits*
  - Sfruttano dei bug nel sistema Android per ottenere permessi di Root
  - In questo modo, una applicazione malevola può prendere il controllo del sistema
  - Esempio: (libsysutils, Gingerbreak, RATC, etc.)
  - Tutti per versioni fino alla 2.3.6!
- *Remote Control*
  - Trasformazione dello smartphone in un bot
  - Utilizzato da quasi tutti i malware!
  - Vengono utilizzati dei commandi HTTP per pilotare i bot da alcuni server specifici
  - A volte la comunicazione è criptata (es. PjApps)

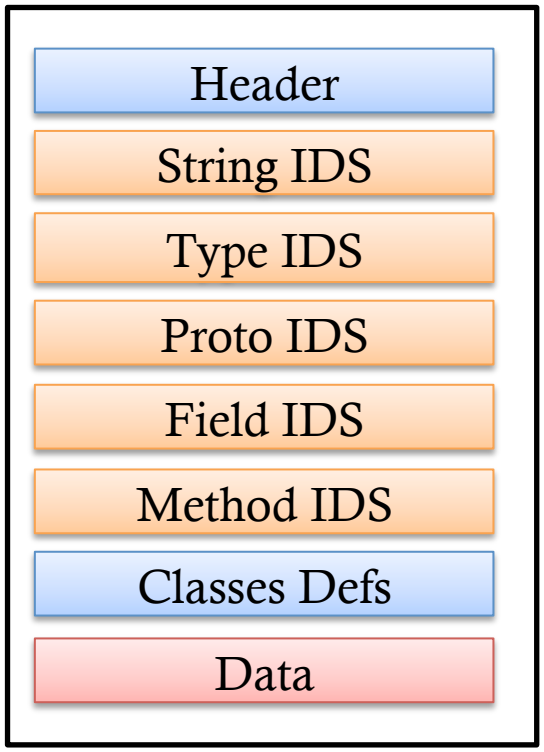
Tipi di Attacco (2)

- *Financial Charge*
  - Esiste una funzione chiamata sendTextMessage (attivabile con permesso), capace di mandare sms senza che l'utente possa controllarlo
  - Ovviamente, i numeri contattati sono premium a costo altissimo…
  - A volte, viene proprio effettuata una "sottoscrizione" a servizi premium
    - Il malware è programmato per mandare determinate "risposte" al servizio che ne comportano la sottoscrizione
- *Information Stealing*
  - Molte informazioni possono essere di interesse per un attaccante…
  - Numeri di telefono (rubrica)
  - SMS
  - Indirizzi mail
  - Credenziali Utente

Offuscamento

- L'offuscamento è un insieme di strategie per rendere il codice meno leggibile quando viene disassemblato
- Agiscono principalmente sul Dalvik Bytecode (cioè, il contenuto del classes.dex)
- Ma anche sul Manifest.xsml
- Queste tecniche vengono usate per evadere anche gli antivirus che si basano sul bytecode o sul Manifest per individuare gli attacchi
- Per capire al meglio l'offuscamento, è importante mostrare come l'eseguibile Dalvik è **strutturato**

Struttura Classes.dex



Header

Sezione delle stringhe

Sezione dei tipi (formata da stringhe che rappresentano I tipi che verranno usati da metodi, campi e classi)

Sezione dei prototipi (definisce tipo di ritorno, parametri e nome dei metodi)

Sezione dei campi (attributi di classe)

Sezione dei metodi (definisce le informazioni del metodo, come ad esempio il suo bytecode)

Sezione delle classi (definisce le informazioni sulla classe, come il loro tipo)

Dati (il codice che viene eseguito)

Tecniche di Offuscamento

- **Trivial**
  - Rinomina I nomi delle classi, dei metodi, dei campi
  - Agisce sulla sezione delle stringhe
- **String Encryption**
  - Le stringhe usate nell'eseguibile (ad esempio le variabili) vengono decriptate a runtime con dei metodi aggiuntivi
  - La sezione dei metodi viene espansa, e le stringhe nella loro sezione vengono rinominate
- **Reflection**
  - Tutte le invocazioni di metodi (e a volte anche di campi) vengono sostituite da delle chiamate "introspettive" che hanno lo stesso effetto
  - Si ricorre alla Java Reflection API
  - Agisce sulla sezione dei metodi e sui dati
- **Class Encryption**
  - Le classi vengono criptate
  - Vengono poi decriptate a runtime e caricate dinamicamente da altre classi
  - Tale operazione modifica PESANTEMENTE il file eseguibile

Offuscamento - Esempio

Trivial

Originale

Class descriptor : ' **Lcom/tt/yy/GoogleReaderServices$ReadingBinder**;
'

Access flags : 0x0001 (PUBLIC)
Superclass : ' Landroid/os/Binder;'

Offuscato

Class descriptor : ' **Lcom/tt/yy/b**;'
Access flags : 0x0001 (PUBLIC)
Superclass : ' Landroid/os/Binder;'

Reflection

Originale

this.counter=10;

Offuscato

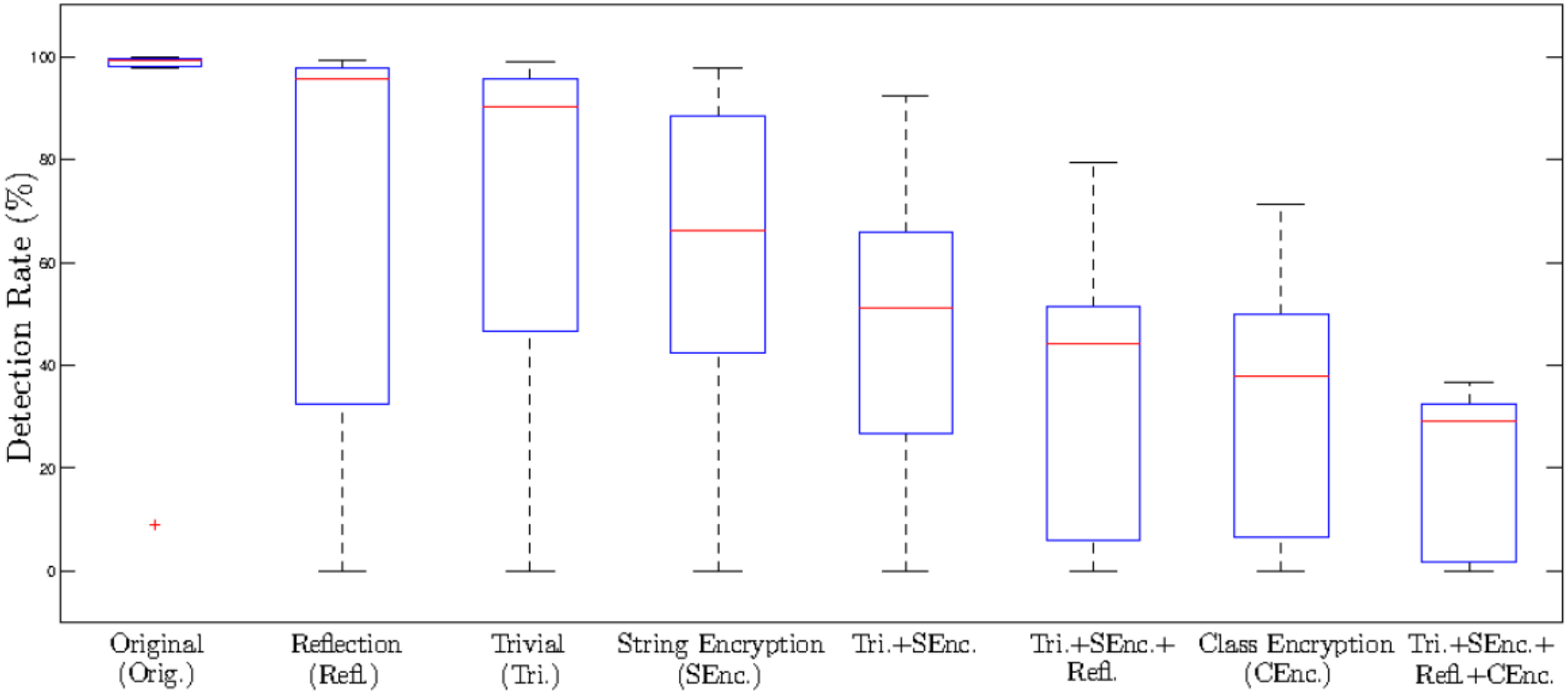
Field
myCounter=MyClass.class.getDeclaredField("counter");
myCounter.set(this,10);

Ricerca sulla Sicurezza

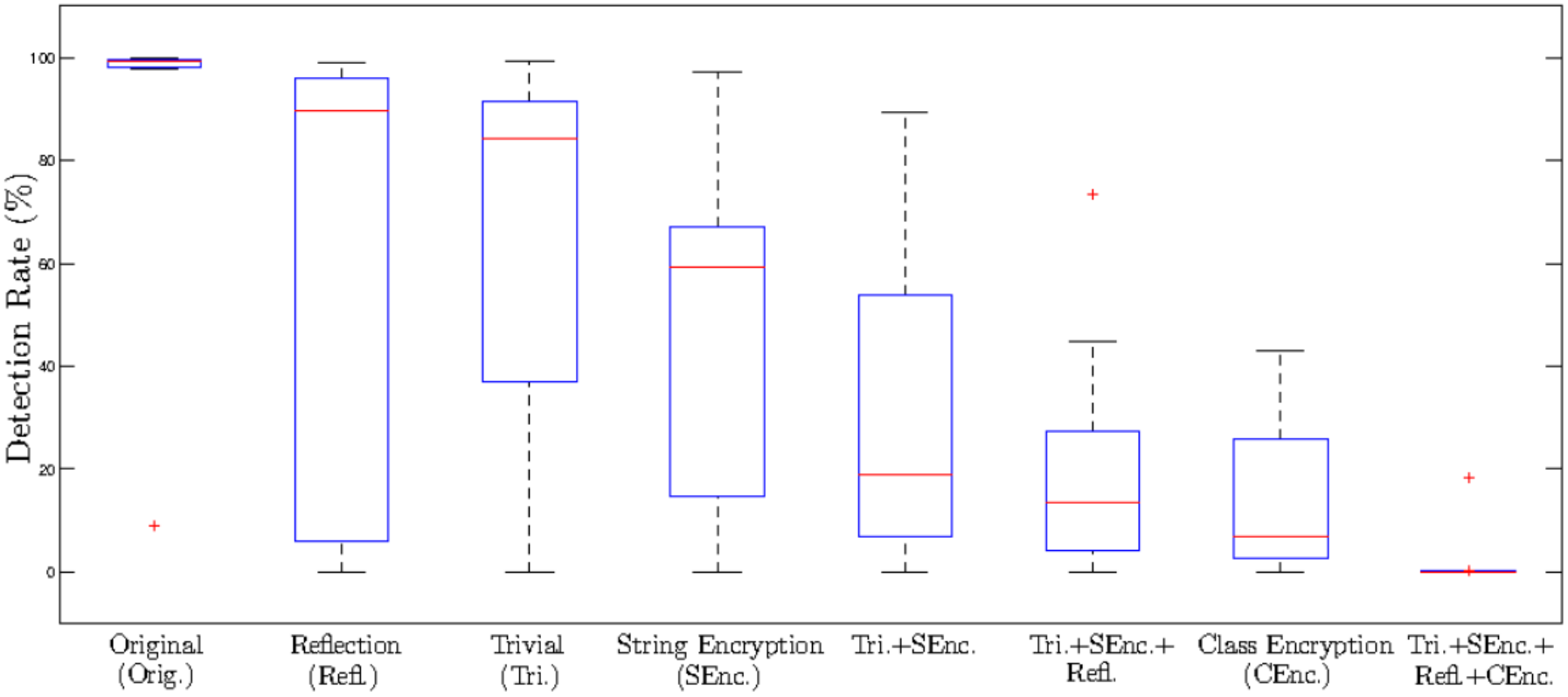
Analisi Antivirus

- Un primo lavoro è stato quello di valutare l'efficienza di diverse tecniche di offuscamento contro un gran numero di Antivirus
- Abbiamo considerato più di 1500 malware
- E li abbiamo offuscati usando **7 tecniche diverse**
- Inoltre, abbiamo considerato tali offuscamenti sotto **tre diversi scenari**
  - Offuscamento di APK
  - Offuscamento di APK, inclusi eventuali assets (in particolare, le librerie native)
  - Offuscamento di APK, influesi eventuali assets ed entry points (classi di "partenza")
- Abbiamo testato I campioni offuscati contro **13 AntiVirus (AV)**
- I grafici rappresentano quanti antivirus sono capaci di rilevare un determinato offuscamento
- Il limite inferiore della scatola indica la **detection rate massima per il 25% degli AV**
- La linea rossa indica **la detection rate massima per il 50% degli AV**
- Il bordo superiore della scatola indica **la detection rate massima per il 75% degli AV**

Analisi Antivirus (Detection APK)

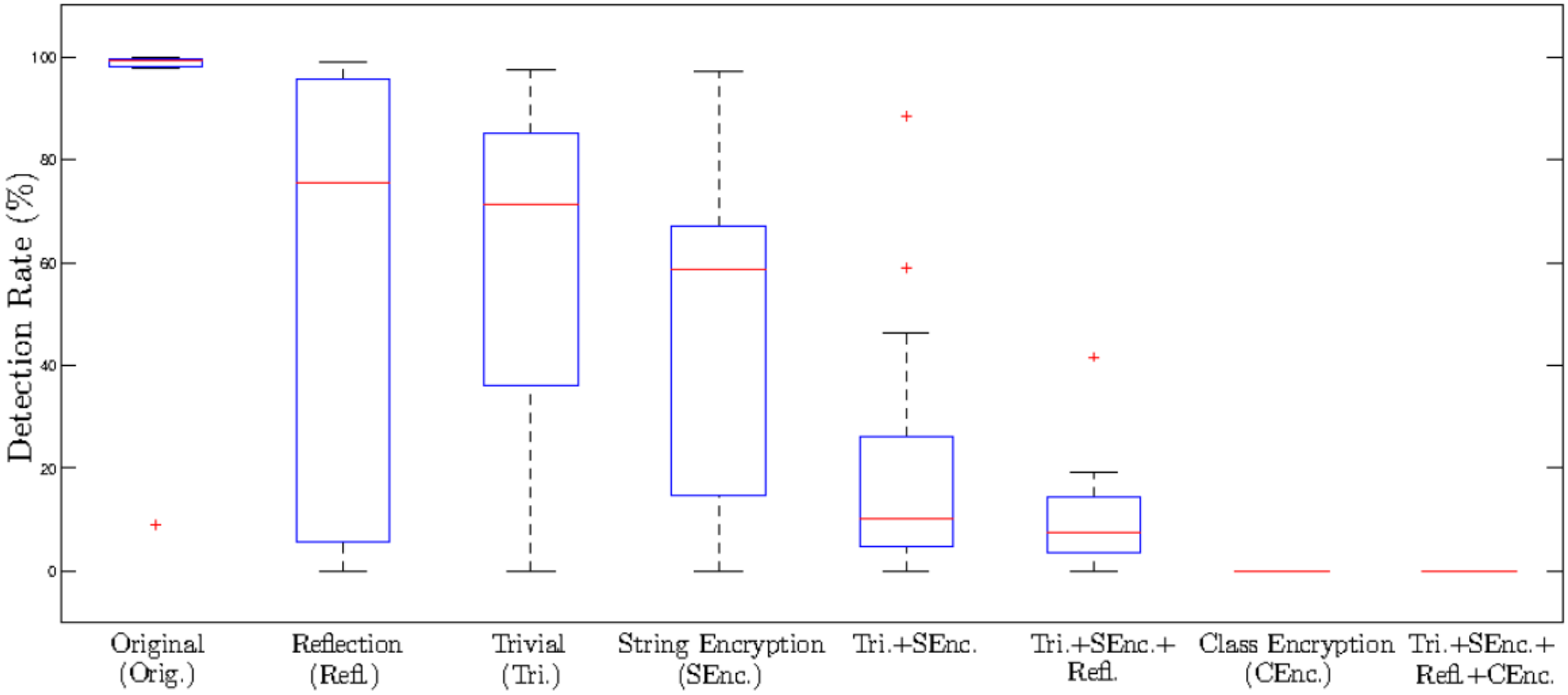


Analisi Antivirus (Assets Criptati)



Le tecniche più complesse sono capaci di azzerare la detection quando gli Antivirus non possono controllare gli assets!

Analisi Antivirus (Assets Ed Entry Points Criptati)



Le stringhe delle classi Entry Point vengono spesso controllate!

Evoluzione Antivirus Nel Tempo

Table 4 – Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (First set of Malware Samples) – See text for explanation of notation.

Antivirus	DroidDream	Geinimi	FakePlayer
AVG	Tri.→ *→ **CEnc.+AE**	Tri.→ *→ *+**SEnc.**	Tri.→ *→ *+**AE**
Symantec	Naive→ Tri.→ **CEnc.**	Tri.→ *→ **CEnc**	Tri.→ *→ **SEnc**
ESET	AE→ Tri.+*→ **Refl.**	SEnc.→ *→ **Tri.**+*+**Refl.**	Tri.→ *→ *+**SEnc**
Kaspersky	AE → *+StrEnc.→ **Tri.**+*	Tri. → *→ *+**SEnc.**	Tri. → *→ *+**CEnc.**
Trend M.	AE → *+Tri.→ **CEnc.+AE**	Tri.→ *→ *+**SEnc.**	Nai.→ *→ **Tri+EP**

Table 5 – Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (Second set of Malware Samples) – See text for explanation of notation.

Antivirus	Bgserv	BaseBridge	Plankton
AVG	Tri.→ *→ **SEnc.+AE**	Tri.→ *→ **SEnc.+AE**	Tri.→ *→ *+**EP**
Symantec	Tri.+SEnc.→ *→ **CEnc.**	Nai.→ SEnc→ *	Nai.→ *→ **SEnc**
ESET	Tri.→ *→ **CEnc.**	AE.→ *+SEnc.→ *	Nai.→ Tri.+SEnc.→ *+**Refl.**
Kaspersky	Tri+SEnc. → *→ **CEnc.**	Tri.+SEnc. → *→ *+**AE**	Nai. → *→ **Tri.+SEnc.**
Trend M.	Nai. → Tri.→ **CEnc.+AE**	AE→ *+FR→ *+**Tri.**	Nai.→ *→ **Tri+SEnc.**

Tecniche più complesse sono necessarie per evadere un Antivirus rispetto ad un anno fa…ma non basta ancora!

Analisi Tools di Ricerca

Analizzati e Raggruppati 28 Tools di Analisi Applicazioni Android

Funzionalità multiple: Detection dI Malware, Permessi, Vulnerabilità, Analisi Comportamentale

Analisi Statica e Dinamica

Diverse Tecniche di Analisi (Decompilazione, Call Graph ed Api Stringh Matching)

Obiettivo: Offuscare in modo tale da evadere sia l'analisi statica che quella dinamica

		Code Analysis		Requirements		
	Tool	Static	Dynamic	Decomp.	CG	APISM
General	AndroGuard [2]	✗			✗	✗
	DroidScope [3]		✗			
	FlowDroid [19]	✗		✗	✗	
	Mobile-Sandbox [20]	✗	✗			✗
	TaintDroid [18]		✗			
MW	DroidMOSS [21]	✗				
	DroidRanger [4]	✗	✗		✗	✗
	RiskRanker [22]	✗		✗	✗	✗
Permissions	COPES [27]	✗				✗
	Pegasus [28]	✗		✗	✗	✗
	PermissionWatcher [23]					
	Permission Risk Signals [24]					
	Prob. Generative Models [25]					
	Stowaway [5]	✗				✗
	VetDroid [29]		✗			
	WHYPER [26]					
ICC vulnerabilities	CHEX [31]	✗			✗	
	ComDroid [6]	✗			✗	✗
	DIDFAIL [36]	✗		✗	✗	✗
	DroidChecker [32]	✗		✗	✗	✗
	Epicc [34]	✗		✗	✗	✗
	IccTA [35]	✗		✗	✗	✗
	SCanDroid [30]	✗		✗	✗	✗
	Woodpecker [33]	✗			✗	✗
Behaviour	AppIntent [39]	✗	✗	✗	✗	
	AppProfiler [38]	✗		✗		✗
	CopperDroid [7]		✗			
	ProfileDroid [37]	✗	✗			✗

Creazione di Offuscatori

Contro gli AV avevamo usato offuscatori commerciali: non avevamo controllo di come si offuscava!

Creare un Offuscatore-> Pieno controllo del Dalvik Bytecode

VM registers (**VERY HARD** – Superare l'Android Verifier!)

Classes.dex Sections

Android Manifest

Listing 2. Indirect static field and local array access without revealing the object type. Code in lines 4–6 is replaced with semantically equivalent code from lines 8–15.

```
1  const/4 v2, #int 1
2  new-array v1, v2, [Lj/l/String;
3
4  sget-object v0, Lexmpl/Main;.aField:Lj/l/String;
5  const/4 v2, #int 0
6  aput-object v0, v1, v2
7
8  const-class v9, Lexmpl/Main;
9  const-string v10, "aField"
10 invoke-virtual/range {v9, v10}, Lj/l/Class;.getDeclaredField
       :(Lj/l/String;)Lj/l/reflect/Field;
11 move-result-object v8
12 invoke-virtual/range {v8, v9}, Lj/l/reflect/Field;.get:(Lj/l
       /Object;)Lj/l/Object;
13 move-result-object v0
14 const/4 v2, #int 0
15 invoke-static {v1, v2, v0}, Lj/l/reflect/Array;.set:(Lj/l/
       Object;ILj/l/Object;)V
```

Abbiamo offuscato applicazioni famose(e.g., What's app, Firefox, Twitter...)

TABLE IV. ARTIFICIAL BENCHMARK RESULTS. VALUES IN SECONDS.

Device	Obf.	Socket	File	Process	Array
Nexus 5 (ART)		0.7756	0.1581	2.4756	0.0127
Nexus 5 (ART)	✗	1.4549	0.9422	2.6515	0.4890
Galaxy Nexus		1.5791	0.1972	1.6423	0.0375
Galaxy Nexus	✗	1.9243	0.8896	2.7481	0.8598

Risultati su Analisi Statica e Dinamica

Tipi Nascosti

Signatures Distrutte

Call Graph Distrutto

Manipolazione Permessi ed Entry Points

Tutti Gli Analizzatori Statici Vengono Evasi dal Nostro Offuscatore

Overtainting (Anti Taint Analysis)

«Inquinamento» di Entry Points

Ritardi «Artificiali»

Detection di Emulazione

TABLE II. RESULTS FOR DYNAMIC ANALYSIS SERVICES.

Vendor	Direct	Sleep	Alarm	EmuDetect	Network	Taints
Anubis	✓	ϟ	✓	ϟ	✓	ϟ
ForeSafe	ϟ	ϟ	ϟ	ϟ	✓	*n. a.*
Mobile Sandbox	✓	✓	✓	ϟ	✓	ϟ
NVISO	✓	ϟ	ϟ	ϟ	✓	ϟ
Tracedroid	✓	ϟ	✓	ϟ	✓	*n. a.*

Create diverse applicazioni con cui abbiamo evaso l'analisi dinamica

Conclusioni

- Android è un settore molto “ricco” dal punto di vista della sicurezza
- E’, infatti, la piattaforma mobile preferita per I malware
- Il loro numero di malware è molto vasto!
- E gli Antivirus commerciali si stanno evolvendo…
- Ma la ricerca scientifica è molto attiva da questo punto di vista
- Numerose tecniche di analisi per applicazioni Android sono state sviluppate
- Noi abbiamo anche mostrato come queste tecniche possano essere deboli e si possano superare
- E’ un settore, perciò, in continua evoluzione!