



Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas, Físico - Química y Naturales
Departamento de Computación

Trabajo Final de la Licenciatura en Ciencias de la
Computación

Confidencialidad de Datos en Aplicaciones Android

Diego Gastaldi

Director: Francisco Bavera

16 de diciembre de 2016

Resumen

Aplicaciones **Android**, tanto maliciosas y mal intencionadas como programadas descuidadamente o mal, pueden filtrar información importante de los usuarios. Para detectar estas filtraciones de información se pueden analizar las aplicaciones con distintos enfoques. Esta trabajo presenta el desarrollo de una herramienta basada en análisis estático para detectar filtraciones de información. Combinando y extendiendo diferentes trabajos previos como *DidFail* [1], *Flowdroid* [2], *Epicc* [3], *Soot* [4], entre otros se obtiene una herramienta que permite realizar el análisis sobre un conjunto de aplicaciones, donde se identifican las posibles comunicaciones inter-aplicación (comunicación entre componentes de diferentes aplicaciones) e intra-aplicación (comunicaciones entre componentes de una misma aplicación), con el fin de detectar filtraciones de información confidencial.

El usuario puede crear un jerarquía de niveles de seguridad y asignárselas a diferentes informaciones que pueden fluir entre las distintas aplicaciones. La herramienta permite configurar una política de seguridad multinivel. Junto con ellas, el usuario puede brindar excepciones, permitiendo a flujos que violen los niveles de seguridad no ser considerado peligroso. Este último punto, permite flexibilidad y que la herramienta pueda ser utilizada en aplicaciones reales.

La herramienta genera información del flujo de los datos en el programa, luego, con la información obtenida se analiza que el flujo de la información no viole la política de seguridad asignada por el usuario (que se respete la jerarquía de niveles de seguridad de los datos). Este último paso se realiza aplicando una implementación del algoritmo de *Jacob Rehof* y *Torben Mogensen* [5].

Este informe describe el método de análisis, implementación y resultados experimentales de la herramienta obtenida.

Índice general

Resumen	I
1. Introducción	1
1.1. Motivación	1
1.2. Contribución	2
1.3. Estructura del Informe	3
2. Conocimientos Previos	4
2.1. Análisis Estático	4
2.2. Information-Flow	5
2.2.1. Niveles de Seguridad	7
2.2.2. Sinks y Sources	8
2.3. Android	9
2.3.1. Introducción a la plataforma Android	10
2.3.2. Modelo de seguridad de Android	14
2.3.3. Android y Malware	14
2.4. Ejemplo Motivador	15
2.5. Herramientas de Análisis Estático	19
2.5.1. Flowdroid	19
2.5.2. Epicc	20
2.5.3. Didfail	20
2.6. Trabajos Relacionados	21
3. Diseño e Implementación	22
3.1. Escenario de ejemplo	24
3.1.1. Información obtenida a través de Didfail	24
3.1.2. Información obtenida a través de archivo de configuración únicamente	25
3.1.3. Información obtenida mediante GUI o archivos de configuración	27
3.1.3.1. Asignación de niveles a métodos	27
3.1.3.2. Excepciones	29
3.2. Etapa 1	30
3.3. Etapa 2	32

3.4. Compilación y uso	35
4. Resultados Experimentales	37
4.1. Descripción de las aplicaciones a analizar	37
4.2. Datos	38
4.2.1. Jerarquía de niveles de seguridad	38
4.2.2. Asignación de niveles	38
4.2.3. Excepciones	39
4.2.4. Flujos de información detectados	39
4.3. Resultados obtenidos	41
5. Limitaciones	43
6. Conclusión y futuros trabajos	45
Bibliografía	47

Capítulo 1

Introducción

Este proyecto consiste en el desarrollo de una herramienta basada en análisis estático de flujo de la información para programas **Android** con el fin de verificar la ausencia de filtraciones (no deseadas) de información confidencial. La herramienta toma como entrada un conjunto de aplicaciones destinadas al sistema operativo **Android**, luego las decompila (es decir, obtiene su código en algún lenguaje específico, como puede ser Java o un lenguaje intermedio), para así, poder analizar el flujo de la información de las aplicaciones y comprobar si estas aplicaciones tiene un comportamiento malicioso o no.

1.1. Motivación

Este trabajo surge a causa de la deficiencia que tienen los repositorios de aplicaciones **Android**, ya que, a pesar de que brindan una manera automática, de costo mínimo, para resolver la distribución o sustitución de código para enormes cantidades de destinatarios, esta técnica también entraña graves riesgos, ya que el software de los repositorios puede comprometer la confidencialidad y/o integridad de los datos de sus numerosos destinatarios, tanto por fallas de programación, como por intenciones maliciosas. Este problema no es nuevo, ni exclusivo de **Android**, por lo que la búsqueda de soluciones para este problema ha dado origen a activas líneas de investigación [6].

Un ejemplo de la explotación de estas deficiencia se da cuando un usuario de un dispositivo (smartphone, tablet, etc) instala un juego para **Android** que filtra toda

la lista de contactos del usuario (obteniendo previamente permiso para hacerlo) a una empresa de marketing mediante el envío de los contactos a otra aplicación con los permisos necesarios para acceder a internet, llevándose a cabo una fuga de información que no fue detectada ni controlada por el sistema.

El problema de la integridad de la información manipulada por aplicaciones **Android** es de gran importancia, dado el intenso uso de esta plataforma. Pero, es muy difícil y costoso realizar las pruebas y la depuración para garantizar la inexistencia de dicho problema. Por ello, es necesario contar con técnicas y herramientas automáticas que verifiquen estáticamente estas propiedades.

La aplicación de análisis estáticos inter e intraprocedural que permitan determinar el flujo de la información permitirá garantizar la confidencialidad e integridad de la información manipulada por programas **Android**. La construcción de un prototipo podrá clarificar la viabilidad, eficiencia y eficacia del uso de análisis estático para la verificación de confidencialidad e integridad de la información en este tipo de aplicaciones.

1.2. Contribución

La herramienta obtenida como resultado de este trabajo combina y extiende las funcionalidades provistas por trabajos previos como **DidFail** [1], **Flowdroid** [2] y **Epicc** [3] añadiendo la posibilidad de asignar niveles de seguridad a cada uno de los métodos que obtienen información del usuario y a cada método que permite enviar dicha información a, por ejemplo, otra aplicación o internet. Esta asignación de niveles de seguridad es, nada más ni nada menos, que la definición de la política de seguridad del usuario. Los niveles de seguridad están relacionados por un orden parcial. Dicho orden también es definido por el usuario. La extensión a las herramientas mencionadas anteriormente utiliza el orden parcial de los niveles y la política de seguridad definida por el usuario para comprobar si la información fluye de manera segura. Es decir, si la información fluye de acuerdo a la política de seguridad definida o se produce alguna una violación (pérdida no deseada de información).

Tanto los niveles asignados a cada método como los flujos que deben ser ignorados (excepciones) pueden ser proporcionados por el usuario de dos maneras diferentes: mediante el uso de una interfaz gráfica o mediante archivos de configuración.

También, se brinda un archivo de configuración para el establecimiento de la jerarquía que relaciona los niveles de seguridad (es un retículo de niveles de seguridad). Ésta debe cumplir con ciertos requisitos que se mencionarán a lo largo del informe.

En cuanto a los resultados, se informa del flujo que produce la violación, en caso que exista, o de la ausencia de estos en caso contrario. Dichos resultados pueden verse de la misma manera que las entradas al sistema, es decir, de manera gráfica o guardado en un archivo.

Además, el usuario tiene la posibilidad de no proveer niveles a los métodos que crea conveniente, permitiendo a la herramienta el cálculo de un nivel adecuado para evitar problemas de seguridad (en caso de ser posible). Estos niveles calculados automáticamente se incluyen a los resultados del análisis.

Por otro lado, se realizaron modificaciones a `Didfail` [1] para realizar los análisis individuales de las aplicaciones de manera concurrente, y se generó una alternativa para los casos en que las aplicaciones ya fueron analizadas individualmente y solo se pretende verificar nuevas asignaciones de niveles de seguridad, ya que la primer parte del análisis es la más costosa en cuanto a tiempos y recursos que utiliza `DidFail`.

La herramienta desarrollada se encuentra disponible bajo licencia *Open Source* en https://github.com/diegogastaldi/security_android.git.

1.3. Estructura del Informe

El resto del informe está organizado como se describe a continuación. El capítulo 2 provee una introducción a los conceptos importantes para este trabajo, como es el análisis estático, **Android** (conceptos relacionados con el tema) y un resumen de las herramientas utilizadas. Además, se presenta un ejemplo motivador. El capítulo 3, describe en detalle el aporte provisto por este trabajo. En el capítulo 4 se describe la implementación de la herramienta desarrollada. En cuanto al capítulo 5, contiene un ejemplo del uso del analizador juntos con los resultados obtenidos. Se discuten las limitaciones del análisis en el capítulo 6, y las conclusiones a las que se llegaron en el capítulo 7.

Capítulo 2

Conocimientos Previos

2.1. Análisis Estático

Se denomina análisis estático al conjunto de técnicas de análisis en el cual el código fuente es analizado sin ser ejecutado. Como contrapunto, el análisis dinámico involucra el estudio del comportamiento de las aplicaciones a través de la ejecución de las mismas en un ambiente determinado (los dispositivos android en nuestro caso).

El análisis estático permite examinar todas las posibles ejecuciones de un programa. Esto es especialmente valioso en el análisis de aspectos de seguridad, ya que los ataques suelen explotar a las aplicaciones de maneras imprevistas y (generalmente) difíciles de *testear*. Pero, predecir el comportamiento del programa sin ejecutarlo no es un problema trivial. Al reducir el problema anterior al Problema de la Parada (*Halting Problem*), es posible demostrar que la búsqueda de todas las maneras posibles de ejecutar cualquier programa arbitrario no trivial es un problema indecidible. Sin embargo, el análisis estático puede proporcionar resultados útiles mediante la aproximación de algunos aspectos de la ejecución real de un programa [7].

Una de las técnicas de análisis estático lleva a cabo el análisis del flujo de datos (*data flow*) en particular, en este trabajo, se focaliza en análisis de Flujo de la Información *Information-Flow*.

2.2. Information-Flow

La protección de la confidencialidad de la información manipulada por los sistemas de computación no es un problema de los últimos años. Por el contrario, es un problema de larga data siendo cada vez más importante. Las prácticas de seguridad estándares, previas al surgimiento de estudios sobre *information-flow* [6], no ofrecían garantías sustanciales de que el comportamiento de extremo a extremo (*end-to-end*) de un sistema informático satisfacía *confidencialidad*. Confidencialidad es una política de seguridad de gran relevancia. Una política de confidencialidad de extremo a extremo afirma que un atacante no puede deducir información (datos) considerados secretos a través de observaciones sobre las salidas del sistema.

Los mecanismos de seguridad convencionales, tales como *control de acceso* y *cifrado* no se refieren directamente a la aplicación de las políticas de flujo de información. Una forma estándar para proteger los datos confidenciales es el control de acceso, el cual requiere de algún privilegio para poder acceder a los archivos u objetos que contienen los datos confidenciales. Las verificaciones de control de acceso no imponen restricciones al acceso a la información, pero no de su propagación y/o divulgación. Es decir, una vez que la información es accedida, el programa que accede a ella puede, por error o malicia, puede transmitir la información ha diferentes destinos. El uso de cifrado para garantizar la confidencialidad tiene la misma limitación, ya que, una vez descifrada la información puede ser liberada sin ningún control. De igual manera, los mecanismos de seguridad tales como la verificación de firmas y escaneo de antivirus tampoco garantizan la conservación de la confidencialidad de la información.

Analizar confidencialidad de un sistema informático es difícil. Los sistemas pueden incluir errores de implementación y diseño. Sumado a esto, los sistemas informáticos modernos son cada vez más complejos, ya que, comúnmente incorporan *hosts* o código no confiable (posiblemente maliciosos) haciendo que la garantía de confidencialidad sea aún más difícil de garantizar.

Para asegurar que la información se utiliza de acuerdo con las políticas de confidencialidad pertinentes, es necesario analizar cómo fluye la información dentro del programa. La certeza de que un sistema es seguro con respecto a la confidencialidad debe surgir de un análisis riguroso que demuestre que el sistema en su conjunto hace cumplir las políticas de confidencialidad de sus usuarios. Este análisis debe

demostrar que la información controlada por una política de confidencialidad no puede fluir a un lugar donde se viola esa política. La *Figura 1* muestra un ejemplo básico indicando un flujo permitido y uno que no lo es (que viola las políticas de seguridad). En la *Figura 1* H representa datos confidenciales (privados) y L datos no confidenciales (públicos). Si *fluye* información privada a pública (información pública depende de datos privadas) entonces el atacante puede deducir información confidencial.

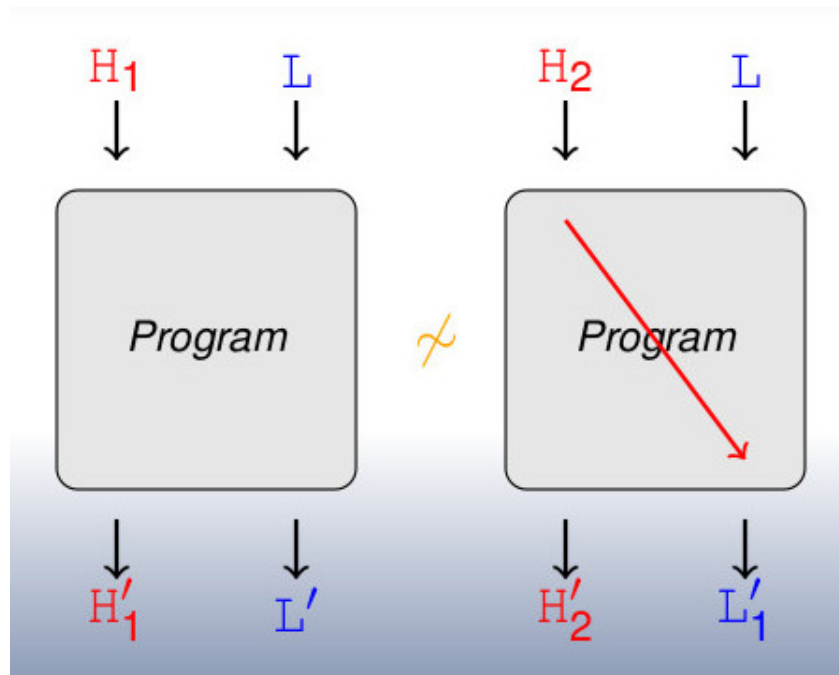


Figura 1: Flujos permitidos y no permitidos.

Las políticas de confidencialidad que deseamos cumplir son, por lo tanto, las políticas de flujo de información y los mecanismos que las hacen cumplir son los controles de flujo de información. En este tipo de enfoques se utilizan técnicas de lenguajes de programación para especificar y hacer cumplir las políticas de flujo de información.

El análisis de confidencialidad e integridad para lenguajes de bajo nivel (*assembly*, *bytecode*) [8–12] en la actualidad tiene un menor desarrollo que el alcanzado para lenguajes de alto nivel (Java, C). Esto se debe, principalmente, a la dificultad de razonar con programas no estructurados. Existen trabajos basados en sistemas de tipos que incluyen un subconjunto bastante extenso de Java *bytecode* [9, 10, 13, 14].

Numerosos trabajos se dirigen a extender el análisis para que soporte *desclasificación* de información [15]. Entre estos podemos nombrar *Intransitive noninterference* [12], *decentralized label model* [16], *relaxing noninterference* [17, 18] y *robust declassification* [19]. En el caso específico del último mencionado, no es un tema cerrado [20] ya que no hay trabajos específicos en cuanto a desclasificación para lenguajes de bajo nivel, como bytecode: para verificar que programas en bytecode satisfacen *robust declassification* es necesario poder verificar que los ataques (código no confiable insertado en determinados puntos del programa) cumple con ciertos requisitos. Por lo tanto, si se quiere contar con una herramienta para verificar la seguridad de los programas que pueda ser utilizada en la práctica es necesario poder verificar que los ataques no violan los requisitos. Actualmente no existen análisis ni herramientas que garanticen *robust declassification* para aplicaciones Android.

Los resultados preliminares sobre técnicas de *information-flow* para bytecode [14, 21] y *robust declassification* para bytecode [19, 22], si bien contribuyen al conocimiento de análisis de flujo de la información, no son técnicas enfocadas a aplicaciones Android. Dado el estado del arte en el área de *information-flow* y *desclasificación* es necesario avanzar en la definición de análisis para aplicaciones Android.

2.2.1. Niveles de Seguridad

El punto de partida en el análisis del flujo de información es la clasificación de las variables del programa en diferentes niveles de seguridad. La forma tradicional de clasificar los datos y las variables de un programa es L (*low*) indicando que son datos de baja seguridad (información pública); y H (*high*) a los datos y variables de alta seguridad (información privada). El objetivo es prevenir que la información privada se filtre de manera incorrecta, es decir, evitar que la información con nivel H pueda ser asignada a las variables con nivel L.

En términos generales, se requiere un retículo de niveles de seguridad para asegurar que la información fluya solamente de niveles menores a niveles mayores o iguales. Por ejemplo, si $L \leq H$, entonces se permitirían los flujos de L a L, de H a H, y de L a H, y no estarían permitidos flujos de H a L.

Claramente es ilegal un flujo explícito donde se le asigna una variable pública el contenido de una privada, pero por el contrario, asignar información pública en variables privadas es perfectamente legal. En el siguiente fragmento de código, la variable `secreto` es H y la variable `fuga` es L.

```
secreto = fuga; // Flujo seguro.  
  
fuga = secreto; // Flujo inseguro.
```

Otro caso, que puede ser considerado peligroso, es cuando de acuerdo a condiciones que involucren información privada se realice una determinada acción, como muestra el ejemplo a continuación:

```
if ((secreto % 2) == 0) fuga = 0; else    fuga = 1;
```

En este fragmento de código se copia el último bit de la variable privada `secreto` en la variable pública `fugas`. Un posible atacante puede inferir información privada observando el valor de la variable pública. Es decir, se filtra información privada.

Otro caso interesante en el uso de niveles de seguridad para garantizar políticas de seguridad esta relacionado con *integridad* en lugar de confidencialidad. En el caso de integridad se analiza que variables (datos) que contienen información (posiblemente) contaminada no fluyan a variables no contaminadas. Se puede modelar esto utilizando un retículo con sus elementos denominados, por ejemplo, `Untainted` \leq `Tainted`. Integridad y confidencialidad son propiedades duales.

2.2.2. Sinks y Sources

Taint analysis es un tipo especial de análisis de flujo de datos que realiza el seguimiento de datos a lo largo del camino de la ejecución del programa. En esta técnica, los datos sensibles son marcados con un tipo (comunmente denominado mancha o tag) en la fuente u origen (*source*), y se propaga a través de todas las rutas de ejecución del programa hasta algún destino *sinks*. La presencia de este mancha en los *sinks* se utiliza para establecer un flujo entre el *source* y el *sink*. Este flujo puede ser utilizado para detectar si existen fugas de datos sensibles desde la fuente a diferentes destinos. Es decir, si se asocian niveles de seguridad tanto a

los *source* como a los *sinks*, se puede detectar si información importante o privada puede estar llegando a lugares públicos o lugares no deseados.

Los *sources* y *sinks* son componentes muy importantes en los flujos de información. Estos son los recursos mediante los cuales las aplicaciones leen u obtienen sus datos (*sources*), para luego tratarlos según sus objetivos y concluir con el envío de estos a otros recursos denominados *sinks*. Estos recursos pueden ser en muchos casos externos a las aplicaciones.

Dichos componentes generan dependencias que van desde determinados *sources* a determinados *sinks*, por lo que al asignarle niveles de seguridad a ambos (véase *sección 2.2.1*) permitiría controlar o conocer los casos en los que se producen flujos ilegales de información.

Ejemplos de *sources* pueden ser el identificador de un teléfono, los contactos, las fotos y la ubicación de los dispositivos móviles, entre otros; y por otro lado, ejemplos de *sinks* incluyen internet, mensajes de texto y archivos, entre otros.

2.3. Android

El sistema operativo **Android** domina el mercado de dispositivos móviles, pero las aplicaciones desarrolladas para **Android** se han enfrentado a algunos problemas de seguridad de gran impacto. Entre estos problemas, a tomado gran relevancia las vulnerabilidades que provocan fugas de datos sensibles.

Todos los sistemas operativos modernos, incluido **Android**, utilizan algún mecanismo de control de acceso para proteger los datos de posibles lecturas o modificaciones por usuarios no autorizados. Sin embargo, controlar el acceso es una medida insuficiente para supervisar la propagación de la información después que la misma ha sido accedida por una aplicación. Similarmente, la criptografía ofrece una fuerte garantía de preservar la confidencialidad pero el costo de realizar computaciones triviales con datos encriptados es muy costoso. Ninguno de estos dos enfoques provee una solución completa para proteger la confidencialidad e integridad.

Un enfoque complementario consiste en analizar y regular el flujo de la información en el sistema para prevenir que se filtre datos privados a lugares no autorizados.

En las aplicaciones **Android** surge un nuevo aspecto a analizar para garantizar la confidencialidad e integridad de los datos: el mecanismo de comunicación entre aplicaciones. En el middleware de **Android**, los *intents* (mensajes entre aplicaciones) son el principal medio de comunicación entre aplicaciones.

Un *intent* puede incluir un destinatario, una acción y, posiblemente, otros datos. Si ningún destinatario es designado en un *intent* (denominado *intent implícito*), entonces **Android** trata de determinar un receptor adecuado, los cuales son las aplicaciones que declaran en su archivo de configuración (*manifest*) que pueden realizar la acción especificada por el *intent*. Si hay varias aplicaciones capaces de atender el *intent*, **Android** solicita al usuario que seleccione la aplicación que atienda el requerimiento. Cabe destacar que el usuario puede designar la aplicación por defecto que procese todos los *intents* similares. Sin embargo, una aplicación maliciosa puede engañar al usuario mediante el uso de un nombre confuso. También, un usuario poco atento podría no dar mucha importancia a la elección. Cuando una aplicación maliciosa recibe un mensaje que fue pensado para otra aplicación, el usuario está ante un ataque de secuestro (de *intent*).

Además de la comunicación entre aplicaciones, los *intents* también se utilizan para la comunicación intra-aplicación. Es decir, entre los diferentes componentes de una sola aplicación. El uso de *intents implícitos* para la comunicación intra-aplicación ha demostrado ser un error común en el desarrollo de aplicaciones de **Android**. Un componente que utiliza un *intent implícito* para comunicarse con otro componente en la misma aplicación podría ser vulnerable a que otra aplicación intercepte su mensaje. Esto permite a aplicaciones maliciosas secuestrar o espiar en aplicaciones que tienen acceso a información o recursos sensibles.

2.3.1. Introducción a la plataforma Android

Android [23] es un *stack* de software de código abierto creado para una amplia gama de dispositivos, el cual está compuesto por cinco componentes principales:

Kernel de Linux: Provee acceso al hardware.

Código nativo de userspace: Incluyen servicios y bibliotecas de sistemas que se comunican con servicios y drivers de bajo nivel.

Bibliotecas y servicios en Java .

Máquina virtual Dalvik / ART: Provee una capa de abstracción eficiente al sistema operativo.

Aplicaciones Android

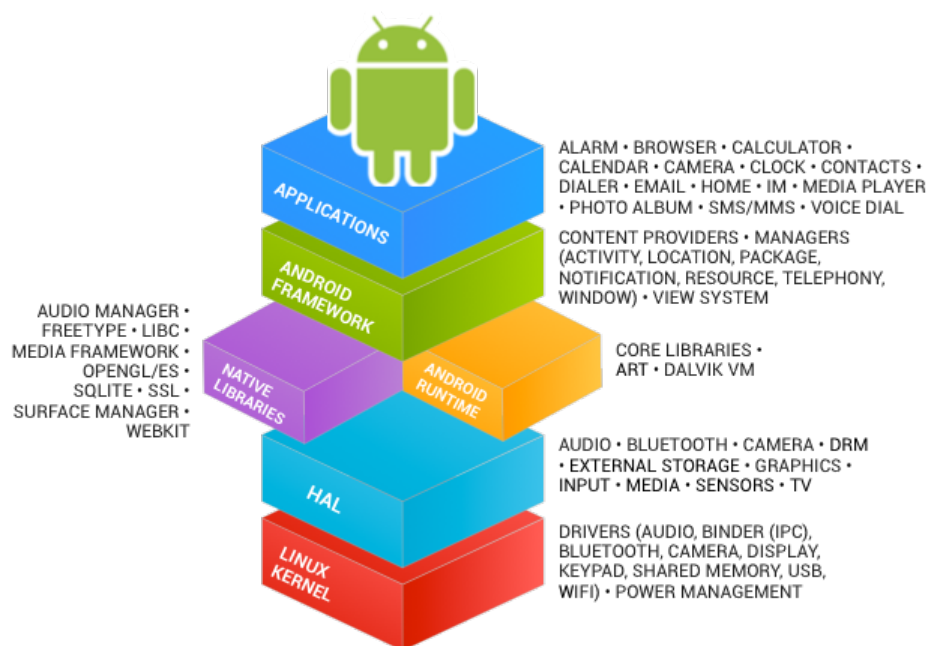


Figura 2: Android Stack.

Android [23] trabaja en Linux, donde cada aplicación utiliza un proceso propio. Aquí los dispositivos tienen un único foco de ejecución principal, que es la aplicación que está visible en la pantalla, pero puede tener varias aplicaciones en un segundo plano, cada una con su propia pila de tareas.

La pila de tareas es la secuencia de ejecución de procesos en **Android**. Ésta está compuesta de actividades que se van apilando según son invocadas, y solo pueden terminar cuando las tareas que tiene encima ya lo hicieron, o cuando el sistema las destruye porque necesita memoria, por lo que tienen que estar preparadas para terminar en cualquier momento. El sistema siempre eliminará la actividad que lleve más tiempo parada. En caso de que se necesitara mucha memoria, si la aplicación no está en el foco, puede ser eliminada por completo a excepción de su actividad principal.

Una de las características principales del diseño en **Android** es la reutilización de componentes entre las aplicaciones, es decir, dos aplicaciones diferentes pueden utilizar una misma componente, aunque éste en otra aplicación, para así evitar la repetición innecesaria de código y su consiguiente ocupación de espacio.

Los componentes de una aplicación son los elementos básicos con los que se construyen. Si bien hay cuatro tipos de éstos, las aplicaciones están compuestas principalmente de actividades: habrá tantas actividades como ventanas distintas tenga la aplicación. Sin embargo, por si solos, los componentes no pueden hacer funcionar una aplicación ya que es necesario que estén comunicados. Dicha comunicación es llevada a cabo a través de *intents*

Los *intents* deben declararse en el archivo llamado *AndroidManifest.xml*.

Los cuatro componentes antes mencionados son:

Actividades: una actividad (o *activity*) es la componente principal encargada de mostrar al usuario la interfaz gráfica. Se define una actividad por cada interfaz del proyecto. Las actividades tienen un ciclo de vida, es decir, pasan por diferentes estados desde que se inician hasta que se destruyen: *activo* cuando la actividad está en ejecución; *pausado* cuando aún se está ejecutando y es visible, pero no es la tarea principal; y *parado* cuando la actividad está detenida, no es visible al usuario y el sistema puede liberar su memoria.

Servicio: los servicios (o *service*) son tareas no visibles que se ejecutan siempre por *debajo*, incluso cuando la actividad asociada no se encuentra en primer plano. Tiene un hilo propio (aunque no pueden iniciar su ejecución solos), lo que permite llevar a cabo cualquier tarea, por más pesada que ésta sea.

Receptores de Mensajes de Distribución: también llamados *broadcast receiver*, son los encargados de reaccionar ante los eventos ocurridos en el dispositivo, ya sean generados por el sistema o por una aplicación externa. No tienen interfaz, pero pueden lanzar una *activity* a través de un evento.

Proveedores de contenidos: (o *content provider*, se encargan de que la aplicación pueda acceder a la información que necesita, siempre que se haya declarado el correspondiente *provider* en el *AndroidManifest*.

Intents: los *intents* son el medio de activación de los componentes (excepto los *content provider*). Contiene los datos que describen la operación que desarrollará el componente a quien va dirigido. Pueden ser explícitos o implícitos (no especifican el componente al que va destinado).

Intent-filters: utilizados únicamente por los intents implícitos. Los *intent-filters* definen (y delimitan) los tipos de intent puede lanzar una actividad y los que

puede recibir un broadcast. Por ejemplo, para un intent que no especifica a que actividad va dirigido, se consulta el intent filter de una de ellas, y si lo satisface, el intent lanzará esa actividad. Se definen en el *AndroidManifest* con la etiqueta `<intent-filter>`. La información que pasan los intents debe estar contenida en la definición del intent filter para que el componente pueda ser activado (o pueda recibirlo en el caso del broadcast). Esta información se compone de tres campos: *Action*, *Data* y *Category*.

El campo Action es una cadena de caracteres que contiene la información del tipo de acción que se llevará a cabo. Las acciones pueden ser dadas por la clase Intent, por una API de Android o definidas por el diseñador.

El campo Data contiene la información del identificador (URI) del dato que se asocia a la acción y del tipo de ese dato. Es importante la coherencia ya que si la acción requiere un dato de tipo texto, un intent con un dato de tipo imagen no podría ser lanzado.

Por último, el campo Category es una cadena de caracteres que contiene información adicional sobre el tipo de componente al que va dirigido el intent.

AndroidManifest: este fichero es un documento `xml` en el que se declaran los elementos de la aplicación, así como sus restricciones, permisos, procesos, acceso a datos e interacciones con elementos de otras aplicaciones. Cada elemento se declara con una etiqueta única. No debe confundirse este documento con el `xml` asociado a cada actividad. Los elementos gráficos y distribución de la pantalla serán definidos para cada actividad dentro de su `xml`, pero no en el *AndroidManifest*.

Cada aplicación es comprimida en un paquete **APK**, lo que permite su distribución e instalación. Estos paquetes contienen una colección de archivos, recursos, permisos, entre otras cosas, como muestra la siguiente figura.

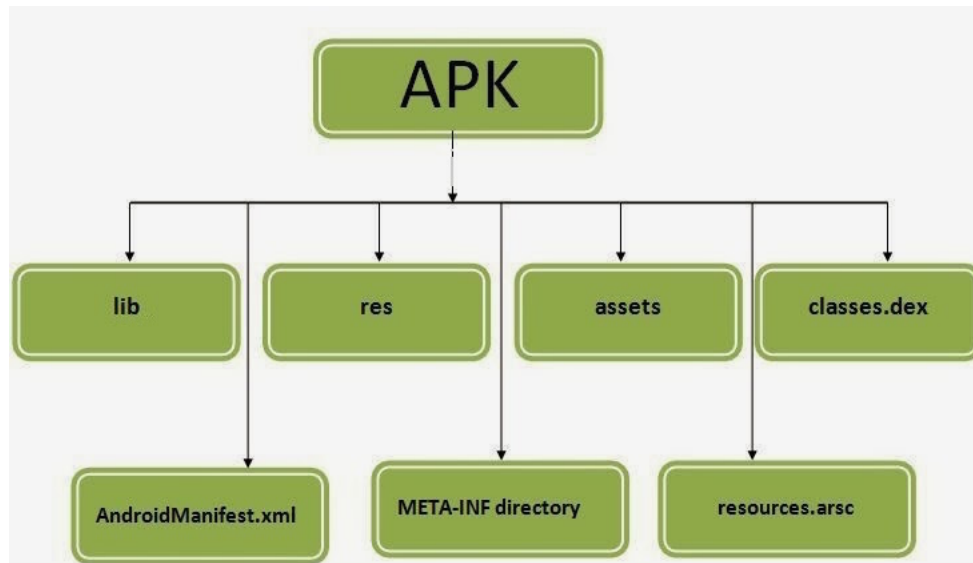


Figura 3: Estructura de paquete Apk de Android.

2.3.2. Modelo de seguridad de Android

Android utiliza dos métodos complementarios de permisos:

Bajo nivel: Permisos de usuarios y grupos de Linux, conocidos como *sandbox* de Android. Estos permisos generan restricciones entre usuarios evitando que interactúen directamente entre sí o que un usuario acceda a los archivos de otros. Desde el punto de vista de las aplicaciones, en la mayoría de los casos cada una de ellas pertenecen a usuarios de Linux diferentes.

Alto nivel: Permisos que las aplicaciones requieren que se les proporcionen para poder efectuar acciones específicas, lo cual regula el acceso de las aplicaciones a los recursos de hardware.

2.3.3. Android y Malware

Malware [24] es el nombre genérico que se le da al software cuyo propósito resulta en causar un perjuicio al usuario (robo de información, falta de disponibilidad en los equipos de la información, gastos no autorizados). Existen diferentes tipos de Malware como virus, troyanos y gusanos.

En el caso del sistema operativo **Android**, este tipo de software logra sus propósitos de diferentes puntos:

Vulnerabilidad del kernel de Linux: Sin bien son relativamente escasos y pueden hacerse obsoletos con una actualización (raramente se producen) son un punto a tener en cuenta.

Abuso de confianza: Cuando la aplicación solicita los permisos para realizar el perjuicio al usuario y éste se los proporciona.

Restricciones por compromiso de diseño: Las tarjetas SD son ejemplo de éstos dado que tienen formato **FAT** y no implementan permisos.

Aplicaciones vulnerables: Ciertas aplicaciones con permisos necesarios para realizar sus funciones puede ser utilizadas por Malware a través de sus vulnerabilidades.

Como defensa ante el Malware se pueden utilizar tanto el análisis estático, como el análisis dinámico como se describió antes en este informe.

2.4. Ejemplo Motivador

La comunicación entre las aplicaciones es muy común y necesario para cumplir con sus objetivos. Esta comunicación la llevan a cabo principalmente mediante el uso de intents. Un ejemplo de ello es una foto, la cual puede fluir a través de diferentes aplicaciones: es sacada por la cámara y almacenada, luego es editada por alguna aplicación de edición para posteriormente ser compartida a través de alguna red social.

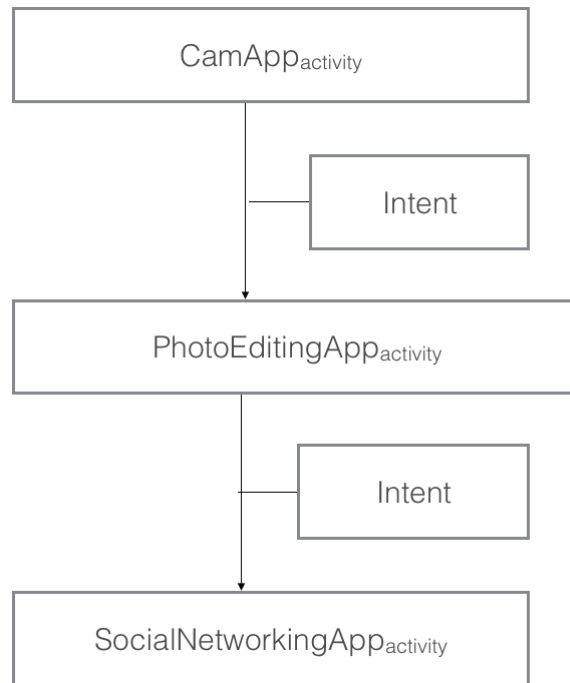


Figura 4: Ejemplo de comunicación entre aplicaciones.

En la siguiente imagen se muestra un ejemplo donde información sensible puede fluir desde un source a un sink, y en su *viaje*, puede pasar por múltiples aplicaciones que pueden ver y modificar su información:

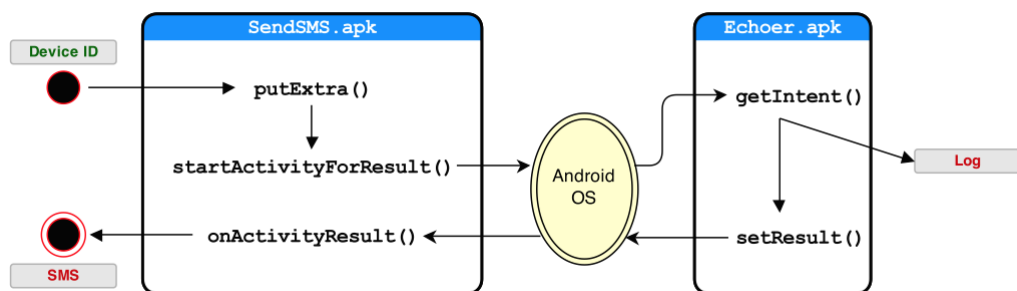


Figura 5: Ejemplo de flujo de información desde un source a un sink.

Aquí la aplicación `SendSms` obtiene `Device Id` (Figura 6:(1)), lo guarda en un intent (Figura 6:(2)) y luego lo envía mediante el método `startActivityForResult` (Figura 6:(3)) para iniciar una nueva actividad. Dicho intent es implícito, es decir, no tiene un destinatario preestablecido, por lo que el sistema operativo se encarga de comprobar que aplicaciones pueden manejarlo (mediante la comprobación de sus archivos manifest). En este ejemplo, la aplicación elegida fue `Echoer.apk`

debido que su manifest cumplía con los requisitos (*Figura 10:(4)*). Ésta aplicación recibe el intent, lo guarda en un campo de la clase `MainActivity` (*Figura 8:(5)*), y luego de que se oprima el botón `button1`, la información que tenía el intent es enviada de vuelta a la aplicación `SendSms` (*Figura 9:(6)*), para que este último envíe el mensaje (*Figura 7:(7)*).

En el escenario descrito, el source (`deviceId`), es información privada del dispositivo, puede llegar a dos sinks diferentes: uno de ellos es el `sms` saliente y el otro sink es el `Log`, y de acuerdo a los niveles que se le asignen a estos últimos, los flujos puede o no producir una violación de seguridad.

Las *Figura 8:(5)* y *Figura 9:(6)* muestran como fluye la información al `Log` escribiendo el contenido del intent recibido.

```
public class Button1Listener implements OnClickListener {
    private final MainActivity act;
    public Button1Listener(MainActivity parentActivity) {
        this.act = parentActivity;
    }
    public void onClick(View arg0) {
        Intent i = new Intent(Intent.ACTION_SEND);
        i.setType("text/plain");
        TelephonyManager tManager = (TelephonyManager) this.act.getSystemService(
(1) Context.TELEPHONY_SERVICE);
        String uid = tManager.getDeviceId(); // SOURCE
        i.putExtra("secret", uid);(2) // write sensitive data to Intent
(3) this.act.startActivityForResult(i, 0); // outgoing Intent
    }
}
```

Figura 6: SendSMS.button1listener.java.

```

public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button1 = (Button) findViewById(R.id.button1);
        button1.setOnClickListener(new Button1Listener(this));
    }
    ...
    ...
    protected void onActivityResult(int requestCode, int resultCode, Intent data) { //
        incoming Intent Result
    (7) sendSMSMessage(data.getExtras().getString("secret"));
    }
    protected void sendSMSMessage(String message) {
        SmsManager smsManager = SmsManager.getDefault();
        smsManager.sendTextMessage("1234567890", null, message, null, null); // SINK
    }
}

```

Figura 7: SendSMS.MainActivity.java.

```

public class MainActivity extends Activity {
    Intent i;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button1 = (Button) findViewById(R.id.button1);
        button1.setOnClickListener(new Button1Listener(this));
    }
    protected void onResume()
    {
        super.onResume();
        i = getIntent(); // read data received in Intent from the caller
    (5) Bundle extras = i.getExtras();
        Log.i("Data received in Echoer: ", extras.getString("secret")); // SINK
    }
}

```

Figura 8: Echoer.MainActivity.java.

```

public class Button1Listener implements OnClickListener {
    private final MainActivity act;
    public Button1Listener(MainActivity parentActivity) {
        this.act = parentActivity;
    }
    public void onClick(View arg0) {
    (6) this.act.setResult(0, this.act.i); // send received data back to the caller
        this.act.finish();
    }
}

```

Figura 9: Echoer.button1listener.java.

```
...
...
<activity
    android:name="echoer.MainActivity"
    android:label="@string/app_name" >
    <intent-filter>      (4)
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
...
...
```

Figura 10: *AndroidManifest.xml* in *Echoer.apk*.

En este ejemplo se representa lo que sucede en muchas aplicaciones, ya sea, por mala intención o por descuidos: la aplicación `SendSms` quiere comunicarse con otro componente de sí misma, enviarle información para que éste lo trate según corresponda, para lo cual manda un intent con dicha información. A continuación, `Echoer` recibe el intent, filtra la información, y luego envía otro intent para que la aplicación inicial siga con su normal funcionamiento. Aquí `SendSms` no sabe que su información fue vista y tratada por otra aplicación.

2.5. Herramientas de Análisis Estático

La herramienta detallada en este informe se construyó sobre `Didfail`, la cual a su vez utiliza `Flowdroid` y `Epicc`.

2.5.1. Flowdroid

`Flowdroid` [2] es una herramienta de análisis estático para aplicaciones `Android`, de código abierto. La cual reduce el programa a una representación intermedia que modela el ciclo de vida de las aplicaciones de `Android`.

Analizar dichas aplicaciones es más complicado que analizar un programa en `Java` porque estos corren sin el framework de `Android`, tienen un simple punto de entrada (el método `main`). En cambio, las aplicaciones `Android`, pueden tener múltiples puntos de entradas, llamados implícitamente por el framework. `Flowdroid` resuelve este problema creando un método llamado `dummymain()`, el cual emula el ciclo

de vida de las aplicaciones incluyendo todas las llamadas implícitas que pueden ocurrir.

Flowdroid puede detectar precisamente solo flujos de datos intraprocedurales, ya que los interprocedurales incluyen intents, invocación a servicios, entre otros.

2.5.2. **Epicc**

Epicc [3] analiza la comunicación entre componentes de manera precisa y efectiva, por lo que se podría decir que es un complemento a lo realizado por **Flowdroid**.

Epicc identifica propiedades (tales como action, category and data MIME type) de los intents que pueden ser enviados y recibidos por los diferentes componentes.

2.5.3. **Didfail**

DidFail (*Droid Intent Data Flow Analysis for Information Leakage*) [1] usa el análisis estático para detectar potenciales fugas de información sensible entre un conjunto de aplicaciones **Android**. **DidFail** combina **FlowDroid** y **Epicc** para realizar un seguimiento de los flujos de datos tanto entre componentes como dentro de cada componente de un conjunto de aplicaciones. **DidFail** tiene dos fases en el análisis:

- Dado un conjunto de aplicaciones, primero determina el flujo de datos individualmente por cada una, y las condiciones en las que estos son posibles.
- Luego, basándose en los resultados de la primer fase, enumera los flujos de datos potencialmente peligrosos habilitados por las aplicaciones en su conjunto.

En los capítulos siguientes se presenta como a partir de la salida que proporciona **Didfail**, y mediante modificaciones en dicha herramienta, se le dan niveles a los métodos involucrados en los flujos, se chequean si se produce violaciones de seguridad de acuerdo al orden establecido entre los niveles, ignorando aquellos métodos que son considerados excepciones, y las opciones para introducir dicha información.

2.6. Trabajos Relacionados

Como antecedentes de trabajos relacionados con esta idea podemos nombrar DidFail [1], Flowdroid [2], DroidSafe [25], PScout [26], TaintDroid [27] y IccTA [28].

TaintDroid [8] does real-time taint tracking to dynamically detect data leaks, including flows that traverse multiple apps. The most similar work to DidFail is IccTA [9], 1 which was developed at roughly the same time as DidFail. Like DidFail, IccTA statically analyzes app sets to detect flows of sensitive data. IccTA uses a single-phase approach that runs the full analysis monolithically, as opposed to DidFail's compositional two-phase approach with fast install-time analysis. IccTA is more precise than the current version of DidFail. Because of its greater context sensitivity, it does less overestimation of tainted data reaching sinks.

Capítulo 3

Diseño e Implementación

En esta Capítulo se describe la herramienta desarrollada. Como se explicó previamente el objetivo de la misma es determinar si las aplicaciones **Android** violan la política de seguridad del usuario. Es decir, permite verificar si las aplicaciones filtran o no información confidencial. En la *Figura 11* se muestra los módulos que conforman la herramienta con las entradas y salidas de cada uno de los módulos.

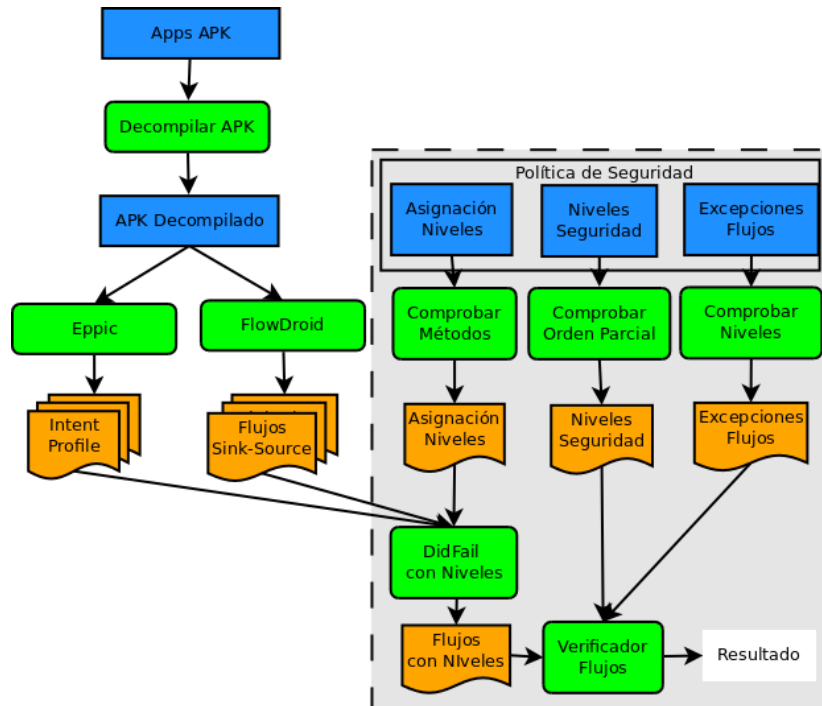


Figura 11: La Herramienta Desarrollada.

Los recuadros *azules* representan datos de entrada provistos por el usuario (**apk** y la política de seguridad). Los recuadros redondeados *verdes* representan módulos

de la herramienta que procesan las entradas y retornan una salida. Los recuadros ondulados *naranja* representan datos de salida de determinados módulos y a la vez son datos de entrada de otros módulos. En el recuadro gris se puede visualizar lo desarrollado en este trabajo.

En resumidas palabras la herramienta toma las salidas proporcionadas por **Epicc** y **FlowDroid**, junto con la descripción de la *política de seguridad* del usuario y las procesa **Didfail con Niveles** (esta es una versión de **Didfail** con extensiones y modificaciones realizadas en este trabajo). El resultado de este último paso consiste en todos los posibles flujos de *sinks* a *sources* de la aplicación de entrada, donde cada *sink* y *source* tendrá asociado el nivel de seguridad definido por el usuario. El último paso identifica si existe algún flujo de información que viola la política de seguridad (o informa la ausencia de dichas violaciones), y además, brinda una asignación de niveles a los *sources* y *sinks* que no fueron considerados críticos por el usuario (es decir, que el usuario no definió un nivel de seguridad porque no los consideró importantes desde el punto de vista de la seguridad).

Cabe notar que la política de seguridad esta conformada por el retículo de niveles de seguridad, la asignación de niveles a *sinks* y *sources* relevantes para la seguridad del usuario y los flujos excepcionales permitidos. En este trabajo se definió el mecanismo para que el usuario pueda explicitar su política de seguridad, se implementaron *validadores* para garantizar que esten definidos correctamente cada uno de sus componentes (en la *Figura 11: Comprobar Métodos, Comprobar Orden Parcial y Comprobar Validadores*). Estos *validadores* también generan una representación de la política de seguridad adecuada para ser manipulada por los restantes módulos.

El análisis realizado por la herramienta puede considerarse dividido en dos etapas: la primera consiste en la recopilación de toda la información necesaria y la generación de los flujos de información de la aplicación (hasta el módulo **DidFail con Niveles**); y la segunda, a partir de la anterior, que verifica generar que los flujos de información no violen la política de seguridad establecida (el módulo **Verificador Flujos**).

Las extensiones y modificaciones a **DidFail** (**DidFail con Niveles**) fueron realizados usando el lenguaje de programación **Python**. Se modificaron diferentes módulos de ésta herramienta y se crearon otros para la implementación de las nuevas funcionalidades.

A continuación se detalla cada uno de los módulos y etapas de la herramienta desarrollada en este trabajo.

3.1. Escenario de ejemplo

3.1.1. Información obtenida a través de Didfail

Un ejemplo de los escenarios a analizar por la herramienta es el que muestra la imagen que sigue (*Figura 11*), donde el componente 1 envía datos al componente 2, este ultimo los recibe y envía otros como respuesta. Por otro lado, el componente 3 interactúa con el componente 2 de manera similar. Vale aclarar que estos componentes pueden o no ser parte de una misma aplicación, lo cual no cambia el resultado en el análisis.

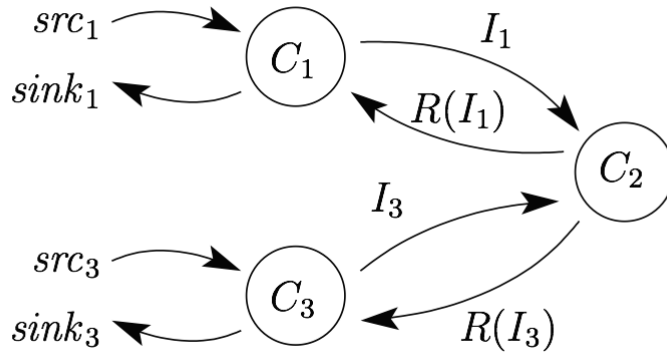


Figura 11: $R(I_i)$ denota la respuesta al intent I_i .

En el escenario de la *Figura 11*, Didfail determina que la información puede fluir de C1 a C2, de C2 a C1, de C3 a C2 y de C2 a C3 de manera directa, pero también pueden suceder flujos de C1 a C3 y de C3 a C1, en el caso de que C2 al recibir la información de, por ejemplo, C1, la guarde en algún campo y posteriormente la envíe como respuesta a C3, luego de recibir un intent proveniente de este último.

Estos flujos son originalmente representados de a pares, donde el componente izquierdo representa el source del flujo (quien envía el intent) y el componente

derecho representa el sink del flujo (quien recibe el intent). Por ende, el resultado seria (source1, sink3), (source3, sink1), (source1, sink1) y (source3, sink3).

A partir de las modificaciones que afectaron a Didfail, los source y sink tienen un campo extra llamado level, que permite, una vez calculado el flujo, comparar los niveles y ver si se corresponden con el orden entre ellos. Un ejemplo concreto de estos se puede observar en la *Figura 12*, la cual muestra un sink con sus correspondientes posibles source (las aplicaciones que generaron esta salida son aplicaciones de prueba llamadas Echoer, SendSms y WriteFile: véase *Capítulo 4*). Aquí el formato esta dado por los diccionarios de **Python**.

```
### 'Sink(sink='Sink: <android.util.Log: int i(java.lang.String,java.lang.String)>', level='Public')': ###  
[Src(src='Src: <android.os.Bundle: java.lang.String getString(java.lang.String)>', level='BUNDLE'),  
Src(src='Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>', level='Private'),  
Src(src='Src: <android.location.Location: double getLongitude()>', level='Private'),  
Src(src='Src: <android.location.Location: double getLatitude()>', level='Private'),  
Src(src='Src: <android.location.LocationManager: android.location.Location  
getLastKnownLocation(java.lang.String)>', level='Private')]
```

Figura 12: Ejemplo de resultados de Didfail.

3.1.2. Información obtenida a través de archivo de configuración únicamente

Otra información necesaria para efectuar el análisis son los niveles de seguridad y el orden que los relaciona, el cual debe ser un orden parcial como requisito excluyente. Un ejemplo de esto se puede ver en la *Figura 13*.

Aquí no es necesario dar las relaciones transitivas (por ejemplo Public - Private) ni tampoco las relaciones reflexivas (Public - Public), las cuales deben estar presentes para cumplir con el requisito de orden parcial pero, para hacer el trabajo del usuario mas simple, las relaciones reflexivas y transitivas son calculadas automáticamente.

Los motivos por el que la relación entre los niveles debe ser de orden parcial son:

Reflexividad: La reflexividad, como indica su definición, cualquier elemento del orden se precede a si mismo. Esto es necesario ya que al momento de comprobar el cumplimiento o no de un flujo de información, si tanto el source como el sink tienen el mismo nivel de seguridad, no hay violación. Si la relación no fuese reflexiva, y por ejemplo, la relación “Public - Public” no esta presente, la existencia de un flujo entre métodos que tienen el nivel “Public” asignado será considerado una violación de seguridad.

Transitividad: Esta propiedad es requerida ya que, al permitirse muchos niveles distintos, se puede dar que exista, por ejemplo, la relación “Public - SemiPrivate” y “SemiPrivate - Private” donde, si la relación “Public - Private” no forma parte del orden, un flujo que vaya desde el nivel “Public” al nivel “Private” sería considerado una violación de seguridad. Esto es un problema ya que sabemos que “Public” precede en el orden a “SemiPrivate” y este ultimo precede a “Private”, por lo que el flujo “Public - Private” no debe ser una violación. Esto se soluciona garantizando que el orden entre los niveles sea transitivo.

Antisimetría: Esta propiedad evita ciclos en el orden, lo cual es claramente necesario debido a que en caso de no cumplirse, todos los niveles que participan en un ciclo (por la transitividad) podrían ser reemplazado por uno solo sin modificar los resultados de los análisis que los usen.

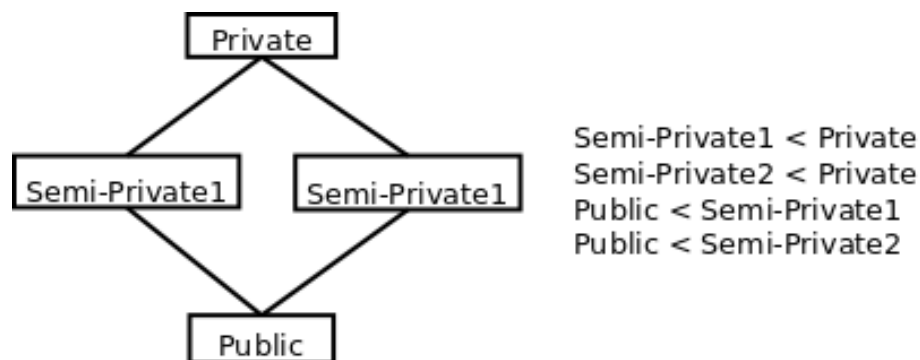


Figura 13: Ejemplo de orden parcial entre niveles de seguridad.

Para brindar los niveles y sus relaciones, el usuario debe modificar el archivo *security-levels.txt*. Aquí debe respetar la siguiente sintaxis:

Comentarios: Las líneas que comienzan con un “#” son ignoradas.

Relaciones: Las relaciones son vistas como dos textos separados por un \leq , siendo los espacios ignorados. Aquí el string que se encuentra a la izquierda de \leq representa al nivel que precede al nivel representado por el string a la derecha del símbolo.

Línea Errónea: Una línea es considerada errónea cuando no contiene el símbolo \leq o cuando tanto a la derecha como a la izquierda de \leq se encuentra un string vacío.

3.1.3. Información obtenida mediante GUI o archivos de configuración

Las entradas que deben proporcionarse a la herramienta que caben en esta clasificación corresponden a la asignación de niveles a métodos y las excepciones.

3.1.3.1. Asignación de niveles a métodos

La asignación de niveles a métodos podrían ser dado por el usuario de dos maneras: mediante la GUI o mediante el archivo de configuración *assign-levels.txt*. En el primer caso la interfaz provee campos de entrada de texto para ser completados con el nombre de la categoría a la que pertenece el método y el nivel a asignarle; en el caso del archivo de configuración, se da la categoría, seguida de una flecha y concluyendo con el nivel a asignar.

Vale aclarar que la categoría es un nombre que se le da a un conjunto de métodos que obtienen o manejan información similar. Por ello es que todos los métodos de una categoría van a tener el mismo nivel. Por ejemplo, los métodos que obtienen tanto la latitud (*getLatitude()*) como la longitud (*getLongitude()*) corresponden a la misma categoría (ACCESS FINE LOCATION) y por ende tendrán el mismo nivel.

Las categorías y sus métodos dependen de la API de android que se este utilizando.

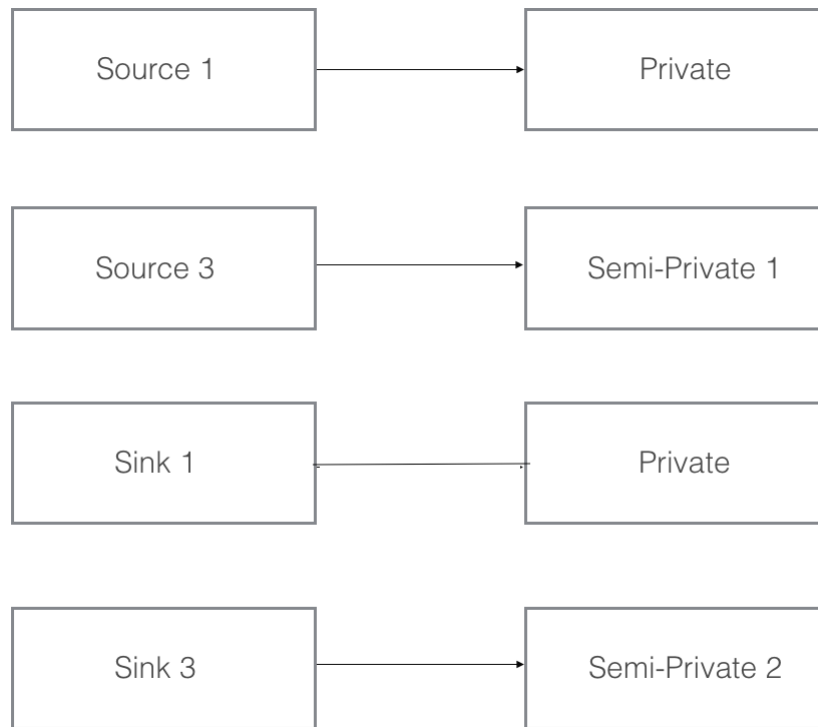


Figura 14: Ejemplo de asignación de niveles de seguridad a sources y sinks.

En la *Figura 14* se muestra un ejemplo de asignación de niveles a métodos donde al método `source1` se le asigna el nivel privado, lo cual se interpreta de la siguiente manera: la información que obtiene la ejecución del método `source1` es información privada que, de acuerdo al orden de los niveles anteriores, no debe llegar a ser argumento de alguno de los métodos con un nivel menor, como son *Public*, *Semi-private1* o *Semi-private2*.

En cuanto a la interpretación de la asignación a `sink1` corresponde a que la información que tiene el método `sink1` como argumento, luego de la ejecución de éste, llega a lugares considerados "Private".

Un ejemplo concreto puede ser, que dado el método `getLatitud()`, el cual obtiene la ubicación del dispositivo considerada normalmente como información privada, se le asigna el nivel *Private*. Por otro lado, la información que es enviada via sms (mediante el método `sendSms()`) normalmente es considerada pública ya que puede llegar a distintas personas. Teniendo en cuenta este escenario, si existe el flujo (`getLatitud`, `sendSms`), existe una violación de seguridad donde información privada arriba a lugares públicos.

3.1.3.2. Excepciones

En el caso de las excepciones, pueden ser:

source1 a sink3

Esto significa que si Didfail detecta un flujo de *source1* a *sink3*, sin importar los niveles que estos tengan asignado, el flujo es ignorado, y no provoca violaciones de seguridad.

Un ejemplo concreto del uso de excepciones que se puede mencionar es el caso en el que una aplicación dada requiere que el usuario se identifique antes de proveerle sus funcionalidades. Para ello el usuario introduce su contraseña, la cual es considerada información privada. En el caso de que la contraseña introducida sea errónea, la aplicación responderá indicando el fallo y posiblemente pidiendo el reingreso de ésta. Aquí surge un inconveniente, donde un componente que recibe información privada, la trata y responde a, posiblemente, un componente considerado público, como por ejemplo, la ventana informando que la contraseña es incorrecta. Si bien la ventana podría contener la contraseña ingresada, lo cual sería claramente una fuga de información, también puede responder simplemente el mensaje de error o éxito. Para situaciones como estas, la herramienta le deja la decisión al usuario, dándole la posibilidad de ignorar la violación de seguridad mediante el uso de excepciones.

Durante el análisis, teniendo en cuenta la posibilidad de manejar excepciones, se puede dar el caso en el que un método que participe en alguna excepción no tenga nivel asignado por el usuario y deba ser calculado por la herramienta. Aquí, si durante el análisis se encuentra el flujo indicado por la excepción, éste no va a limitar la asignación de un nivel al método, es decir, si por ejemplo, el flujo va de *privado* a *variable1* (donde *privado* es el nivel supremo del orden de los niveles y *variable1* es el método que no tiene nivel asignado), al momento de calcular el nivel del método *variable1* no se tendrá en cuenta dicho flujo, es decir, no debe ser mayor o igual a *privado*.

Una vez brindadas las excepciones, la herramienta corrobora que los métodos participantes de la excepción existan en la API de la versión de android con la que se está trabajando.

En este caso, el usuario debe modificar el archivo *exceptions.txt* en caso de que se esté trabajando mediante el uso de archivos de configuración o agregando excepciones a través de la interfaz gráfica. En el caso del archivo, la sintaxis es similar a la mencionada anteriormente, pero con la diferencia de que el símbolo que separa los string es una flecha (\rightarrow) y que a los lados de dicha flecha se colocan métodos.

A modo de ejemplo, la *Figura 15* muestra una excepción donde, si la información que obtiene el método *getDeviceId()* puede ser parte de los argumentos del método *i(java.lang.String,java.lang.String)*, no será considerado violación de seguridad, sin importar los niveles de estos.

Si bien las excepciones podrían haber sido implementadas para que trabajen con categorías en lugar de que lo hagan con métodos directamente, haciendo el trabajo del usuario mas simple e intuitivo, tiene como desventaja que al dar una excepción de una categoría A a una categoría B, en realidad se estarían considerando muchas excepciones (todas las posibles combinaciones de los métodos pertenecientes a la categoría A, con todos los métodos de la categoría B). Por ello es que la excepciones trabajan con métodos, lo que le permite ser mas específicas y manejables.

```
<android.telephony.TelephonyManager: java.lang.String getDeviceId()> ->  
<android.util.Log: int i(java.lang.String,java.lang.String)>
```

Figura 15: Ejemplo de una definición de excepción.

3.2. Etapa 1

En esta etapa, se recolecta toda la información necesaria para efectuar el análisis, la cual puede ser provista por dos vías: mediante la interfaz de usuario o mediante archivos de configuración, dependiendo de la preferencias del usuario.

Una vez que se cuenta con toda la información necesaria, se reemplaza cada método en los flujos por su nivel, de acuerdo a la asignación brindada por el usuario, y en caso de que algún método no tenga una asignación, será considerado variable y su nivel dependerá luego de los demás para evitar la violación del orden de los niveles. Esto es, puede darse el caso de que exista un nivel, el cual se le asigne al método variable y luego el chequeo no de error. O por el contrario, puede ocurrir que no exista dicho nivel y por lo tanto el sistema informará la violación.

En el caso de los niveles de seguridad, se le agregan todos las relaciones necesarias para que el orden resultante sea un orden parcial. De igual manera, dicha relación puede resultar no ser del orden requerido por el no cumplimiento de la antisimetría (requisito para ser orden parcial).

Por otro lado, también se requiere que los métodos participantes en las excepciones deben existir (pertenecer a la API de la versión de **Android** que la herramienta utiliza).

En el caso de las asignaciones de niveles a métodos, los métodos deben cumplir el mismo requisito que los de las excepciones y además, el nivel que se le asigna a cada uno debe pertenecer al orden generado con anterioridad. En cualquier caso que no se cumplan los requisitos antes mencionados, se generará un error y se abortará la ejecución.

En la *Figura 16* se muestra la información necesaria para llevar a cabo la primer etapa del análisis, los chequeos que se realizan para que funcione correctamente y la salida generada en esta etapa.

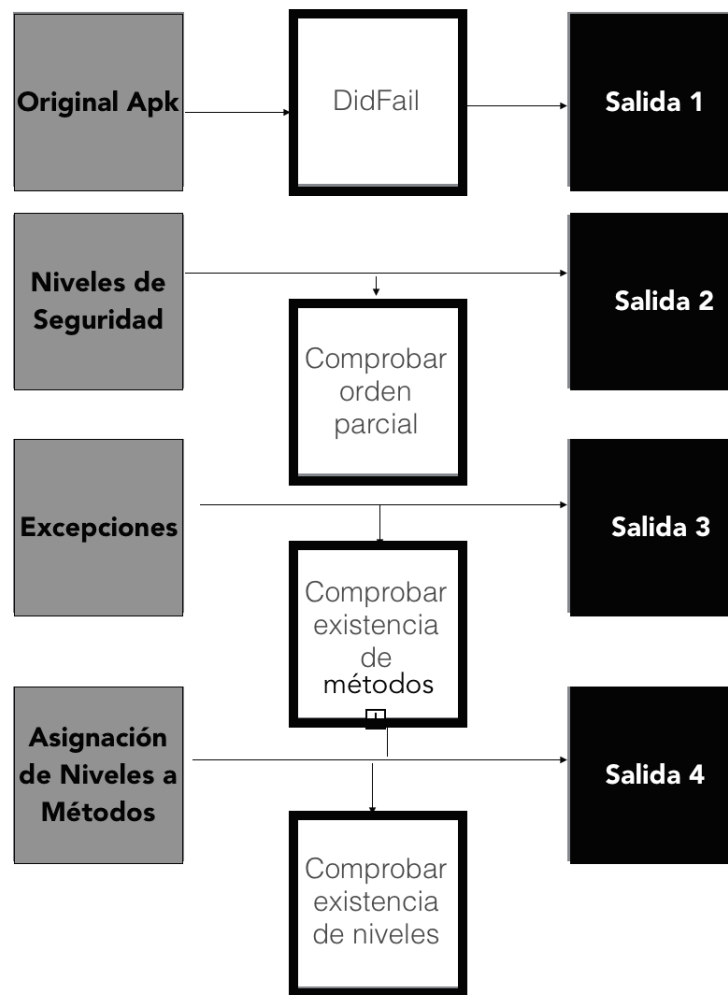


Figura 16: Primer etapa del análisis.

3.3. Etapa 2

Una vez que se cuenta con toda la información necesaria se lleva a cabo la verificación de que los flujos de información detectados no violen la política de seguridad (asignación de niveles de seguridad) definida por el usuario. La idea general de dicha verificación consiste en comprobar que la información no fluya a lugares *más públicos* que su nivel asignado, es decir, que los niveles de los métodos con los que esta información es recopilada no sean menores (precedan en el orden de los

niveles) a los niveles de los métodos donde puede llegar dicha información. Pero, dado que puede suceder que el usuario no asigne niveles de seguridad a todos los métodos, es necesario analizar las dependencias del flujo de información para asignarles el nivel de seguridad que le corresponde a estos métodos. Para luego proceder con la verificación de si se viola o no la política de seguridad.

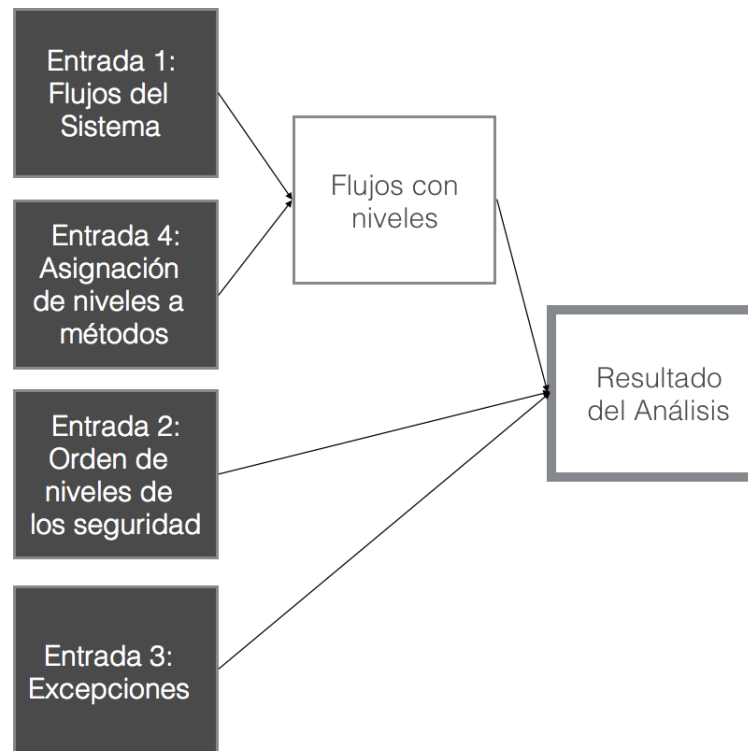


Figura 17: Segunda etapa del análisis.

En la *Figura 17* se puede observar la información necesaria para llevar a cabo esta segunda etapa, las cuales se corresponden con la salida de la *etapa número 1*. Aquí tomando los flujos provenientes de *Didfail* (*Entrada 1*) y las asignaciones de niveles a métodos (*Entrada 4*) genera nuevos flujos a los que les agrega el campo `level` correspondiendo tanto para el `source` como para el `sink`. Luego, a partir de los nuevos flujos generados, sumado a la *Entrada 2* y *3*, se ejecuta una implementación del algoritmo de *Jacob Rehof y Torben Mogensen* [5] a partir del cual se obtiene la salida del análisis. La *Figura 18* muestra el pseudocódigo de dicho algoritmo, el cual chequea si se produce una violación de seguridad comprobando

si en cada flujo de información, sus niveles se corresponden con alguna relación del orden parcial proporcionado (Entrada 2) de niveles. Vale aclarar que previo a cualquier chequeo, se comprueba si el flujo pertenece a una excepción, y en tal caso nunca será considerada violación de seguridad.

```

1. Input
   A finite set  $C = \{\tau_i \leq A_i\}_{i \in I}$  of definite inequalities over  $\Phi = (L, F)$ ,  $F = \{f : L^{a_f} \rightarrow L\}$  with each  $f$  monotone,  $L$  finite lattice,  $Var(C) = \mathcal{V}_m$  for some  $m$ .

2. Initially
    $C := \text{L-NORMALIZE}(C)$ 
    $\rho := \perp_{\mathcal{V}_m \rightarrow L}$ 
    $C_{var} := \{\tau \leq A \in C \mid A \text{ is a variable}\}$ 
   Initialize the lists  $Clist[\alpha]$ , for every distinct variable  $\alpha$  in  $C$ .
   Initialize  $Ilist$  to hold the inequalities in  $C$ .
    $NS := \{\tau \leq \beta \in C_{var} \mid L, \rho \not\models \tau \leq \beta\}$ 

3. Iteration
   while  $NS \neq \emptyset$  do
      $\tau \leq \beta := \text{POP}(NS)$ ;

      $\rho(\beta) := \llbracket \tau \rrbracket \rho \sqcup \rho(\beta)$ ;

     for  $\sigma \leq \gamma \in Clist[\beta]$  do
       if  $\rho, L \not\models \sigma \leq \gamma$ 
       then INSERT( $\sigma \leq \gamma$ )
       else DROP( $\sigma \leq \gamma$ )
     end; (* for *)

   end; (* while *)

4. Output
   If  $L, \rho \models \tau \leq c$  for all  $\tau \leq c \in C_{cnst}$  then output  $\rho$  else FAIL.

```

Figura 18: Algoritmo de Jacob Rehof y Torben Mogensen.

Además del funcionamiento antes mencionado, el algoritmo le asigna niveles a los métodos a los que en la primer etapa del análisis no se les proporcionó uno. Esto lo realiza asignándole el menor nivel del orden a cada uno de éstos métodos y luego comprobando que en todos los flujos en los que éste participa como **sink**, si el nivel actual asignado es el supremo entre su nivel y el del otro método participante del flujo. En caso de ser así, el nivel no es modificado, pero si por el contrario esto no se cumple, al método variable se le asigna el nivel supremo antes mencionado.

Como salida del algoritmo hay dos posibilidades: retorna un mensaje indicando que el chequeo no encontró violaciones de seguridad y la asignación que hizo a los métodos que no tenían niveles para evitar violaciones de seguridad, o bien, informando la violación de seguridad encontrada.

El algoritmo utiliza en su implementación principios tales como pilas, relación de orden, conjunto, entre otros.

A modo de ejemplo, si se tomará como entrada a la *etapa 2* el escenario de la *Figura 11, 13 y 14* de manera tal de que se correspondan con las **Entradas 1, 2 y 4** de la *Figura 17*, por ejemplo, **source1** tiene asignado el nivel **Private**, además, como sabemos que la información obtenida por **source1** puede llegar a **sink3**, y este último tiene nivel **Semi-private3** y dado que el orden de los niveles indica que **Semi-private3** es menor que **Private**, se estaría produciendo una violación de seguridad. Pero, como dicho flujo esta presente en las excepciones, éste es descartado.

Otro caso que puede ocurrir es que la información vaya de un nivel menor a uno mayor o que los niveles de seguridad sean iguales, en los cuales no hay problemas de seguridad. Teniendo en cuenta el ejemplo, este caso es representado por el flujo **source1** \rightarrow **sink1**.

Una última alternativa sería, por ejemplo, el caso en el que el flujo tenga su origen con nivel **Semi-private1** y destino **Semi-private2**. En este caso los niveles son incomparables en el orden, lo que también deriva en una violación de seguridad.

3.4. Compilación y uso

Como se menciona en la *Introducción*, la herramienta desarrollada se encuentra disponible bajo licencia *Open Source* en https://github.com/diegogastaldi/security_android.git.

Para poder utilizar la herramienta que se describe en este informe se requiere de un sistema operativo **Linux** y herramientas tales como **Java**, **gcc**, **wxglade**, entre otras. Todas las dependencias necesarias para compilar y ejecutar la herramienta se pueden descargar e instalar mediante el *script* que se encuentra en el archivo **install.sh**, ubicado en el directorio **root** del proyecto.

Los pasos ha realizar para llevar a cabo la instalación son:

1. Copiar el directorio principal de herramienta en un directorio del sistema:
por ejemplo, **/home/userName/security_android**.
2. Ejecutar el script **install.sh**:
por ejemplo, **/home/userName/security_android./install.sh**

Una vez ejecutado el paso antes descrito, se concluye con la instalación. Para su posterior uso se pueden utilizar los siguientes *scripts* ubicados en el directorio **cert**:

- **run-didfail.sh**: ejecuta el análisis completo (incluye las dos etapas del análisis). Los datos de entrada necesarios para la ejecución son obtenidos desde los archivos:
 - **cert/securityLevels/assign-levels.txt**,
 - **cert/securityLevels/security-levels.txt** y
 - **cert/securityLevels/exceptions.txt**.

La salida se realiza en el archivo **/toyapps/out/out.txt**.

- **run-check-levels-file.sh**: ejecuta solo la segunda etapa del análisis utilizando los archivos antes mencionados.
- **run-check-levels-gui.sh**: ejecuta la segunda etapa del análisis utilizando la interfaz gráfica, tanto para la obtención de la información, como para visualizar los resultados.

Junto con la herramienta se proporcionan tres aplicaciones **Android**. Estas son las aplicaciones utilizadas en los ejemplos del actual informe. Para ejecutar el análisis sobre otras aplicaciones se deben colocar los contenedores **.apk** de dichas aplicaciones en el directorio **toyapps**.

Capítulo 4

Resultados Experimentales

En este capítulo se muestra un ejemplo del análisis realizado por la herramienta desarrollada mediante el uso de tres aplicaciones. Se diferencian las dos alternativas que tienen los usuarios para brindar los datos de entrada a la herramienta.

4.1. Descripción de las aplicaciones a analizar

Las tres aplicaciones a analizar son:

SendSMS.apk: Esta aplicación filtra el *device ID* del usuario a través de un mensaje de texto. Lee el *Device ID*, lo añade a un *intent* utilizando el método *putExtra()*, acto seguido envía el *intent* a través del método *startActivityForResult()* permitiendo a otra aplicación recibirlo y responder. Una vez recibida la respuesta, la aplicación filtra los datos a un mensaje de texto.

Echoer.apk: Esta aplicación recibe un *intent* usando el método *getIntent()*, escribe los datos recibidos en el *Log* y por último, retorna los datos (a quien realizó el *intent*) usando *setResult()*.

WriteFile.apk: Esta aplicación es similar a *SendSMS*, excepto que lee la ubicación del dispositivo y la filtra a un archivo.

4.2. Datos

4.2.1. Jerarquía de niveles de seguridad

En este ejemplo se usaron los niveles de seguridad que muestra la *Figura 19*. En ella se puede observar a modo de comentario la representación gráfica del retículo (el orden parcial) generado.

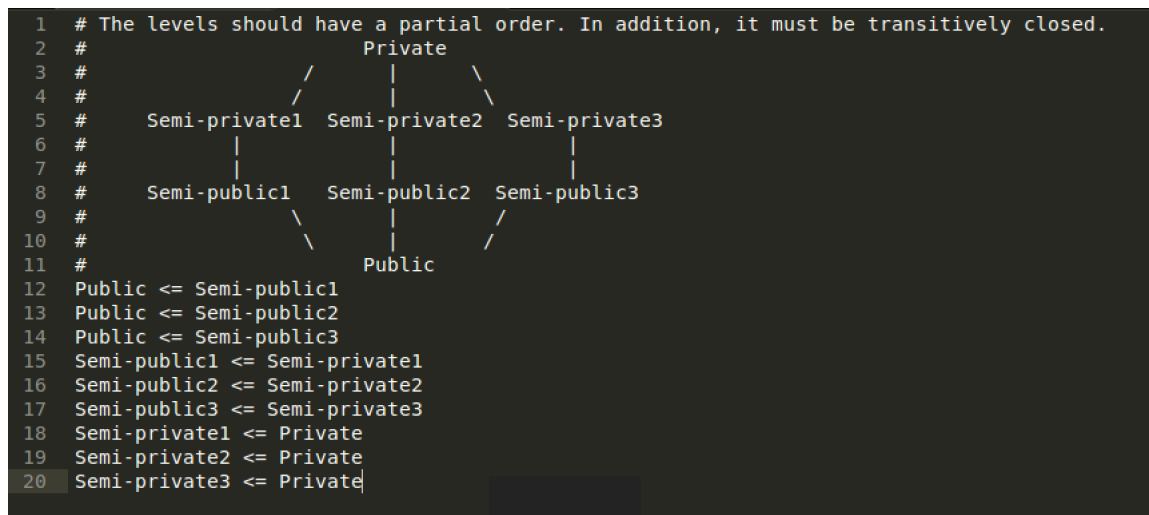


Figura 19: Ejemplo de definición de orden parcial entre niveles de seguridad.

4.2.2. Asignación de niveles

Para mostrar el funcionamiento de la herramienta solo se le asigna niveles a dos categorías como muestra la *Figura 20*, ya que, como se mencionó con anterioridad, no es necesario darle niveles a todas las categorías participantes en los flujos de información, los restantes niveles de seguridad serán inferidos por el análisis. Es decir, en este ejemplo, el usuario considera que la información confidencial de su interés es la asociada a las variables `READ_PHONE_STATE` y `ACCESES_FINE_LOCATION` cuyos niveles de seguridad (confidencialidad) son `Semi-private1` y `Semi-public1`, respectivamente.

```
1 # Format: method -> Security Level
2 # where "Security Level" must be defined in security-levels.txt
3 READ_PHONE_STATE -> Semi-private1
4 ACCESS_FINE_LOCATION -> Semi-public1
5
6
```

Figura 20: Ejemplo de asignación de niveles a categorías de sources y sinks.

4.2.3. Excepciones

La excepción de la Figura 21 hace que se ignore el flujo del *device ID* del usuario al *Log*. Con esta excepción el usuario considera que los flujos de información del *device ID* al *Log* son seguros. Es decir, el usuario permite que se filtre información de un nivel de confidencialidad superior (el *device ID*) a un nivel inferior (el *Log*) porque no considera peligroso este flujo en particular.

```
1 <android.telephony.TelephonyManager: java.lang.String
  getDeviceId()> -> <android.util.Log: int i(java.lang.
  String,java.lang.String)>
2
3
```

Figura 21: Ejemplo de definición de excepción.

4.2.4. Flujos de información detectados

Las Figuras 22 (a, b y c) muestran los flujos de información, los métodos que participan y los niveles de seguridad que tiene asignado cada uno. El formato mostrado se corresponde con una lista de diccionarios (estructura de datos proporcionada por el lenguaje Python), donde cada uno contiene el *source* y el *sink* del flujo, y en estos últimos se especifica el método y el nivel de seguridad correspondiente. En el caso de la Figura 22a se puede observar que, por ejemplo, el análisis detectó un flujo que va del método *android.telephony.TelephonyManager: java.lang.String getDeviceId()* con nivel *semi-private1* al *sink android.util.log: int i(java.lang.String, java.lang.String)* con nivel variable *LOG*. Por lo tanto, éste nivel variable (*LOG*) no podrá tener un nivel menor (no podrá ser más *público*) que el nivel del *source* antes mencionado. Es importante recordar que se denominan

niveles variables a aquellos que no fueron definidos por el usuario y serán inferidos por la herramienta.

En el caso de la *Figura 22b* el *source* o el *sink* corresponde a un *intent*, el cual es identificado por un *id* y en el caso de la *Figura 22c*, ambos componentes del flujo son *intents*.

```
---- Flows with 0 intent(s) -----
[Flow(src=Src(src='Src: <android.os.Bundle: java.lang.String getString(java.
lang.String)>', level='BUNDLE'), app='org.cert.echoer', sink=Sink(sink='Sink: <
android.util.Log: int i(java.lang.String,java.lang.String)>', level='LOG')),
Flow(src=Src(src='Src: <android.telephony.TelephonyManager: java.lang.String
getDeviceId()>', level='Semi-private1'), app='org.cert.sendsms', sink=Sink(
sink='Sink: <android.util.Log: int i(java.lang.String,java.lang.String)>',
level='LOG')),
Flow(src=Src(src='Src: <android.os.Bundle: java.lang.String getString(java.
lang.String)>', level='BUNDLE'), app='org.cert.sendsms', sink=Sink(
sink='Sink: <android.telephony.SmsManager: void sendTextMessage(java.lang.
String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app
.PendingIntent)>', level='SEND_SMS')),
Flow(src=Src(src='Src: <android.location.Location: double getLongitude()>',
level='Semi-public1'), app='org.cert.WriteFile', sink=Sink(sink='Sink: <
android.util.Log: int i(java.lang.String,java.lang.String)>', level='LOG')),
Flow(src=Src(src='Src: <android.location.Location: double getLatitude()>',
level='Semi-public1'), app='org.cert.WriteFile', sink=Sink(sink='Sink: <
android.util.Log: int i(java.lang.String,java.lang.String)>', level='LOG')),
Flow(src=Src(src='Src: <android.location.LocationManager: android.location.
Location getLastKnownLocation(java.lang.String)>', level='Semi-public1'),
app='org.cert.WriteFile', sink=Sink(sink='Sink: <android.util.Log: int i(java.
lang.String,java.lang.String)>', level='LOG')),
Flow(src=Src(src='Src: <android.os.Bundle: java.lang.String getString(java.
lang.String)>', level='BUNDLE'), app='org.cert.WriteFile', sink=Sink(
sink='Sink: <android.util.Log: int i(java.lang.String,java.lang.String)>',
level='LOG')),
Flow(src=Src(src='Src: <android.os.Bundle: java.lang.String getString(java.
lang.String)>', level='BUNDLE'), app='org.cert.WriteFile', sink=Sink(
sink='Sink: <java.io.FileOutputStream: void write(byte[])>',
level='WRITE_EXTERNAL_STORAGE'))]
```

Figura 22a: Flujos que no involucran intents.

```

---- Flows with 1 intent(s) -----
[Flow(src=Intent(tx=('org.cert.WriteFile', None), rx=('org.cert.echoer',
None), intent_id='newField_8'), app=None, sink=Sink(sink='Sink: <android.util.
Log: int i(java.lang.String,java.lang.String)>', level='LOG')),
Flow(src=Intent(tx=('org.cert.sendsms', None), rx=('org.cert.echoer', None),
intent_id='newField_6'), app=None, sink=Sink(sink='Sink: <android.util.Log:
int i(java.lang.String,java.lang.String)>', level='LOG')),
Flow(src=IntentResult(i=Intent(tx=('org.cert.sendsms', None), rx=('org.cert.
echoer', None), intent_id='newField_6')), app=None, sink=Sink(sink='Sink: <
android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.
String,java.lang.String,android.app.PendingIntent,android.app.
PendingIntent)>', level='SEND_SMS')),
Flow(src=Src(src='Src: <android.telephony.TelephonyManager: java.lang.String
getDeviceId()>', level='Semi-privatel'), app=None, sink=Intent(tx=('org.cert.
sendsms', None), rx=('org.cert.echoer', None), intent_id='newField_6')),
Flow(src=IntentResult(i=Intent(tx=('org.cert.WriteFile', None), rx=('org.cert.
echoer', None), intent_id='newField_8')), app=None, sink=Sink(sink='Sink: <
android.util.Log: int i(java.lang.String,java.lang.String)>', level='LOG')),
Flow(src=IntentResult(i=Intent(tx=('org.cert.WriteFile', None), rx=('org.cert.
echoer', None), intent_id='newField_8')), app=None, sink=Sink(sink='Sink: <
java.io.FileOutputStream: void write(byte[])>',
level='WRITE_EXTERNAL_STORAGE')),
Flow(src=Src(src='Src: <android.location.Location: double getLongitude()>',
level='Semi-public1'), app=None, sink=Intent(tx=('org.cert.WriteFile', None),
rx=('org.cert.echoer', None), intent_id='newField_8')),
Flow(src=Src(src='Src: <android.location.Location: double getLatitude()>',
level='Semi-public1'), app=None, sink=Intent(tx=('org.cert.WriteFile', None),
rx=('org.cert.echoer', None), intent_id='newField_8')),
Flow(src=Src(src='Src: <android.location.LocationManager: android.location.
Location getLastKnownLocation(java.lang.String)>', level='Semi-public1'),
app=None, sink=Intent(tx=('org.cert.WriteFile', None), rx=('org.cert.echoer',
None), intent_id='newField_8'))]

```

Figura 22b: Flujos con un intent involucrado.

```

---- Flows with 2 intent(s) -----
[Flow(src=Intent(tx=('org.cert.WriteFile', None), rx=('org.cert.echoer',
None), intent_id='newField_8'), app=None, sink=IntentResult(i=Intent(tx=('org.
cert.WriteFile', None), rx=('org.cert.echoer', None),
intent_id='newField_8'))),
Flow(src=Intent(tx=('org.cert.sendsms', None), rx=('org.cert.echoer', None),
intent_id='newField_6'), app=None, sink=IntentResult(i=Intent(tx=('org.cert.
sendsms', None), rx=('org.cert.echoer', None), intent_id='newField_6')))]

```

Figura

22c: Flujos con dos intents involucrados.

4.3. Resultados obtenidos

Los resultados obtenidos (*Figura 23*) muestran que no se producen flujos de información que viole la política de seguridad (los niveles de seguridad) definidos por el usuario. Además muestra el menor nivel posible, con respecto a los niveles de seguridad, a cada categoría que no tenían un nivel asignado.

La salida también incluye los *intents* lanzados por las aplicaciones. En el conjunto actual de aplicaciones, estos ya fueron utilizados para el análisis y no agrega

información en la salida, pero teniendo en cuenta una posible adición de otra aplicación al conjunto, el usuario tiene la posibilidad de ver el nivel de la información que la nueva aplicación puede obtener (en caso de que su archivo *manifest* así lo permita).

```
1 Applications don't have security problems.
2 Assigned security levels are:
3   {'newField_6': 'Semi-private1'
4    'WRITE_EXTERNAL_STORAGE': 'Semi-public1'
5    'SEND_SMS': 'Semi-private1'
6    'newField_8': 'Semi-public1'
7    'LOG': 'Semi-public1'}
```

Figura 23: Resultado final del análisis realizado por la herramienta.

El nivel asignado a la categoría *Log* merece una aclaración. Como se mencionó en la descripción de las aplicaciones a analizar, al *Log* le llega información desde dos lugares, el *device Id* del usuario con un nivel de seguridad **semi-private1** y la ubicación del dispositivo con nivel **semi-public1**. Para evitar la violación de seguridad, el *Log* debe tener un nivel igual o mayor (más *privado*) que sus *sources*, el cual, sería **semi-private1** o **private**. Sin embargo, lo obtenido por la herramienta es **semi-public1**. Esto ocurre por la excepción dada (*Figura 20*), que ignora el flujo del *device Id* al *log*.

Vale aclarar que si bien en el ejemplo los niveles se refieren a la confidencialidad de la información, la herramienta puede ser utilizada con niveles que se correspondan con la integridad de la misma y así obtener los resultados acorde a estos. Confidencialidad e integridad son dos propiedades duales [29] y pueden verificarse utilizando las mismas técnicas.

Capítulo 5

Limitaciones

La herramienta que se detalla en este informe presenta diversas limitaciones, que pueden clasificarse en dos grupos: aquellas provenientes de las aplicaciones que usa, en especial **FlowDroid** y **Epicc**; y aquellas limitaciones propias de la herramienta. En el primer grupo, una limitación importante es la falta de solidez del análisis producida por las llamadas recursivas y el código nativo, ya que estas no son tratadas de manera eficaz por **Epicc** ni por **FlowDroid**. **FlowDroid** analiza las llamadas que invocan a código nativo de la siguiente manera: si los datos de entrada estaban *contaminados* (forma por la cual **Flowdroid** determina el flujo de la información) antes de la llamada, entonces se determina que todos los argumentos de llamada y cualquier valor de retorno están contaminados. Este manejo no es sólido; no analiza los flujos reales en el código nativo invocado por lo cual la información obtenida puede generar muchos *falsos positivos*. Tampoco son consideradas las fugas de información que involucre *multithread* (hilos de ejecución).

Didfail solo sigue el flujo de la información si la información es transmitida a través de *Intents*. Por ejemplo, si dos aplicaciones se envían información utilizando *campos estáticos* compartidos como medio de comunicación **Didfail** no detectará el flujo de información. Además, **Didfail** también detecta posibles flujos de información que son imposibles que sucedan en realidad.

En el segundo grupo de limitaciones se debe mencionar que el usuario debe dar la definición de los niveles de seguridad de manera correcta. Para esto, el usuario debe tener conocimiento sobre relaciones de orden y configuración de entornos mediante el uso de archivos de configuración. Además, el usuario debe conocer las variables asociadas a cada uno de los recursos del dispositivo **Android** para poder

asociarles los niveles de seguridad deseados. Es necesario resaltar que la cantidad de estas variables es considerable. Tanto la definición de los niveles de seguridad, como la asignación de niveles a cada uno de los recursos del dispositivo móvil, son pasos necesarios para poder utilizar de manera correcta la herramienta.

Cabe resaltar que las limitaciones mencionadas en el segundo grupo se minimizan para el usuario si utiliza la interfaz que provee la herramienta.

Las pruebas de la herramienta fueron realizadas utilizando aplicaciones que podemos denominar de *juguete*, es decir, aplicaciones desarrolladas exclusivamente para probar esta herramienta. Es necesario realizar pruebas a mayor escala, utilizando aplicaciones de uso cotidiano, que pertenezcan al mundo real, para poder comprobar de manera mas certera el rendimiento y la exactitud de los resultados.

Capítulo 6

Conclusión y futuros trabajos

El análisis estático inter e intraprocedural que lleva a cabo el prototipo descrito en este informe permite, no solo, determinar el flujo de la información, sino también, garantizar la confidencialidad e integridad de la información manipulada por programas **Android** siguiendo la configuración (política de seguridad) que establezcan los usuarios de acuerdo a sus necesidades. Este prototipo también permite detectar *sources* y/o *sinks* potencialmente peligrosos; es decir, la información generada por la herramienta brinda a los usuarios información de gran relevancia para determinar niveles de seguridad de *sources* y *sinks* que no fueron considerados en primer instancia por los usuarios (o no fueron considerados sensibles desde el punto de vista de la seguridad) al definir su política de seguridad.

La construcción de este prototipo clarifica la viabilidad, eficiencia y eficacia del uso de análisis estático para la verificación de confidencialidad e integridad de la información.

Es viable porque la implementación así lo demuestra, si bien posee limitaciones y requiere refinamientos tales como la adición de formas en la que el flujo de información puede transmitirse, como por ejemplo los *campos estáticos*, bases de datos **SQLite** y *SharedPreferences*.

En cuanto a la eficiencia, esto es un punto crítico para los dispositivos móviles, ya que, si bien cada vez son más potentes y tienen más recursos, el análisis completo requiere de un gran uso de recursos. Realizar el análisis completo en un dispositivo móvil sería completamente ineficiente y costoso. Como alternativa a ello se puede dividir el análisis en etapas (como las mencionadas en el capítulo de diseño e

implementación de la herramienta) donde en el repositorio de aplicaciones se lleve a cabo el análisis más *pesado* (el cual se corresponde con la etapa número uno) y en el dispositivo solo se realice la segunda etapa del análisis (el cual requiere que se introduzca la política de seguridad y de acuerdo a ella se complete el análisis).

Por otro lado, si bien esta herramienta requiere de refinamiento como se mencionó anteriormente, teniendo en cuenta sus limitaciones y objetivos, realiza un análisis eficaz, detectando los flujos para los cuales esta capacitada para hacerlo y omitiendo aquellos en los que no se tuvieron en cuenta, como por ejemplo, los *campos estáticos*.

Bibliografía

- [1] Jonathan Burket, Lori Flynn, William Klieber, Jonathan Lim, Wei Shen, and William Snively. Making didfail succeed: Enhancing the cert static taint analyzer for android app sets. Technical report, CERT Division, 2015.
- [2] Christian Fritz Eric Bodden Alexandre Bartel Jacques Klein Damien Oceau Patrick McDaniel Steven Arzt, Siegfried Rasthofer and Yves Le Traon. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Technical report, Department of Computer Science and Engineering Pennsylvania State University, 2014.
- [3] S. Jha A. Bartel E. Bodden J. Klein D. Oceau, P. McDaniel and Y. L. Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.
- [4] E. Gagnon L. Hendren P. Lam R. Vallée Rai, P. Co and V. Sundaresan. Soot â a java bytecode optimization framework. *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.
- [5] Jakob Rehof and Torben Mogensen. *Tractable constraints in finite semilattices*. Springer-Verlag, 1996.
- [6] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 2003.
- [7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

- [8] David Pichardie Gilles Barthe and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. *Proc. Of 16th European Symposium on Programming*, 2005.
- [9] N. Kobayashi and K. Shirane. Type-based information flow analysis for a low-level language. 2002.
- [10] Tamara Rezk. *Verification of confidentiality policies for mobile code*. PhD thesis, Universite de Nice and INRIA Sophia Antipolis, 2006.
- [11] Dachuan Yu and Nayeem Islam. A typed assembly language for confidentiality. *European Symposium on Programming*, 2006.
- [12] Amar S. Bhosale. Precise static analysis of taint flow for android application sets. Master's thesis, Software Engineering Institute., 2014.
- [13] A. Basu G. Barthe and T. Rezk. Security types preserving compilation. *Journal of Computer Languages, Systems and Structures.*, 2005.
- [14] Francisco Bavera and Eduardo Bonelli. Typebased information flow analysis for bytecode languages with variable object field policies. 2008.
- [15] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. *CSFW 2005*, 2005.
- [16] Amar Bhosale Limin Jia William Klieber, Lori Flynn and Lujo Bauer. Android taint flow analysis for app sets. *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis - SOAP 2014*, 2014.
- [17] Scott F. Smith Paritosh Shroff and Mark Thober. Dynamic dependency monitoring to secure information flow. *CSF [DBL07]*.
- [18] Andrey Chudnov and David A. Naumann. Information flow monitor inlining. *CSF [DBL10]*.
- [19] Andrei Sabelfeld Andrew C. Myers and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 2006.
- [20] Steve Zdancewic. Challenges for information-flow security. *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence*, 2004.

- [21] F. Bavera and E. Bonelli. ypebased information flow analysis for bytecto-
de languages with variable object field policies. [www.lifia.info.unlp.edu.ar/
eduardo/publications/jvmsLong.pdf](http://www.lifia.info.unlp.edu.ar/eduardo/publications/jvmsLong.pdf), 2008.
- [22] F. Bavera and E. Bonelli. Typebased information flow
analysis for bytecode languages with robust declassification.
http://dc.exa.unrc.edu.ar/docentes/pancho/fbavera/papers/ifbcode_robust.ps http://dc.exa.unrc.edu.ar/docentes/pancho/fbavera/papers/ifbcode_robust.ps
http://dc.exa.unrc.edu.ar/docentes/pancho/fbavera/papers/ifbcode_robust.ps.
- [23] Jorge Cordero Luis Cruz Miguel González Francisco Hernández David Palomero José Rodríguez de Llera Daniel Sanz Mariam Saucedo Pilar Torralbo Manuel Báez, Álvaro Borrego and Álvaro Zapata. *Introducción a Android*.
- [24] Juan Heguiabehere Joaquín Rinaudo. Seguridad y protección de datos en aplicaciones android. Escuela de verano de ciencias informáticas UNRC, 2015.
- [25] Jeff Perkins Limei Gilham Nguyen Nguyen Michael I. Gordon, Deokhwan Kim and Martin Rinard. Information-flow analysis of android applications in droid-safe. Technical report, Massachusetts Institute of Technology, 2015.
- [26] Zhen Huang Kathy Wain Yee Au, Yi Fan Zhou and David Lie. Pscout: Analyzing the android permission specification. Technical report, Dept. of Electrical and Computer Engineering University of Toronto, Canada, 2012.
- [27] B.-G. Chun L. P. Cox J. Jung P. McDaniel W. Enck, P. Gilbert and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *Proceedings of OSDI 2010*), 2010.
- [28] J. Klein Y. L. Traon S. Arzt S. Rasthofer E. Bodden D. Octeau L. Li, A. Bartel and P. McDanie. I know what leaked in your pocket: Uncovering privacy leaks on android apps with static taint analysis. Technical report, 2014.
- [29] K. J. Biba. Considerations for secure computer systems. Technical report, 1977.