

ANÁLISIS DE FLUJOS DE INFORMACIÓN EN APLICACIONES ANDROID

Lina Marcela Jiménez Becerra
Grupo COMIT
Departamento de Ingeniería de Sistemas y Computación
Universidad de Los Andes
Bogotá, Colombia
lm.jimenez12@uniandes.edu.co

Resumen—

El presente trabajo de investigación plantea aplicar técnicas de análisis basadas en control de flujo de información, con el fin de verificar la ausencia de fugas de información en aplicaciones Android, desde su construcción. Puesto que, controlar el acceso y uso de la información, representa una de las principales preocupaciones de seguridad en dichos aplicativos. Un estudio reciente de seguridad en dispositivos móviles, publicado por McAfee[1], revela que en el contexto de aplicativos Android: 80 % reúnen información de la ubicación, 82 % hacen seguimiento de alguna acción en el dispositivo, 57 % registran la forma de uso del celular (mediante Wi-Fi o mediante la red de telefonía), y 36 % conocen información de las cuentas de usuario. Adicionalmente, el informe señala que una aplicación invasiva no necesariamente contiene malware, y que su finalidad no siempre implica fraude; de las aplicaciones que más vulneran la privacidad del usuario, 35 % contienen malware.

Si bien, aplicaciones invasivas no necesariamente implican malware y/o acciones delictivas, el cuestionamiento de fondo es la forma y finalidad con que una aplicación manipula la información del usuario, y qué garantías puede ofrecer el desarrollador para que tal manipulación sea consentida.

CATEGORÍAS Y DESCRIPCIÓN DE TEMÁTICAS

Análisis de flujos de información en aplicaciones.

TERMINOS GENERALES

Técnicas Security-Typed, Técnicas de flujo de información, Técnicas de flujo de datos, Análisis Dinámico, Análisis estático.

PALABRAS CLAVE

Jif, Políticas de seguridad, Flujo de información, Verificación de políticas, Confidencialidad, Fuga de información.

I. INTRODUCCIÓN

En aplicativos Android, el manejo de la información del usuario, es una de las principales preocupaciones de seguridad. Según un estudio reciente de seguridad en dispositivos móviles, publicado por McAfee[1], una importante cantidad

de aplicaciones Android invaden la privacidad del usuario, reuniendo información detallada de su desplazamiento, acciones en el dispositivo, y su vida personal.

Por otro lado, para controlar el acceso a información manipulada por sus aplicaciones, el desarrollador cuenta con los mecanismos de seguridad proveídos por la API de Android, sin embargo, al estar basados en políticas de control de acceso, se limitan a verificar el uso de los recursos del sistema acorde a los privilegios del usuario, lo que suceda con la información una vez sea accedida, está fuera del alcance de este tipo de controles. Al no contar con herramientas de análisis de flujo de información en aplicaciones Android, o al utilizar librerías de terceros, para el desarrollador es difícil verificar el cumplimiento de políticas de confidencialidad e integridad en la aplicación próxima a liberar. Por consiguiente, el desarrollador no tiene cómo asegurar la ausencia de fugas de información en la aplicación.

Si bien, en el campo de aplicativos Android existen diferentes propuestas para detectar fuga de información, en su mayoría se enfocan en precisión y eficiencia del análisis para detectar fugas de datos en aplicaciones de terceros ya implementadas. Estas propuestas no abordan el problema del lado del desarrollador, analizando flujos de información de la aplicación para verificar el cumplimiento de políticas de seguridad.

Ante esto, y con el fin de proveer una herramienta de apoyo al desarrollador, de modo que verifique el cumplimiento de políticas de seguridad en sus aplicaciones, el presente trabajo aborda el problema de fugas de información en aplicaciones Android, analizando flujos de información de la aplicación, mediante técnicas de lenguajes tipados de seguridad.

El Artículo está organizado de la siguiente manera: XXX

II. CONTEXTO

II-A. Aplicaciones Android

En esencia una aplicación Android es una aplicación Java con interfaces descritas en sintaxis XML, cuya ejecución es activada por el framework la API Android.

El framework de Aplicación Android ofrece diferentes funcionalidades de operación del sistema, proporcionando información de los servicios ofrecidos por el teléfono. Por ejemplo, provee información de la ubicación del usuario.

Así pues, una aplicación Android obtiene del framework, clases e interfaces necesarias para implementar sus funcionalidades.

Por otro lado, el SDK, Android Software Development Toolkit, permite compilar la aplicación a una versión ejecutable por dispositivos Android, esto es, código Dalvik bytecode(.dex). Adicionalmente, el SDK genera el APK, Android Application Package, donde empaqueta todo el código de la aplicación, incluyendo el bytecode. El APK es un archivo con extensión .apk, y es el que finalmente se instala en el dispositivo para obtener las funcionalidades de la aplicación.

En lo que respecta a su estructura, una aplicación Android puede integrarse por uno o más de los siguientes componentes: Activities, Services, Content Providers y Broadcast Receivers. Las actividades representan acciones a ejecutar por el usuario, permiten que el usuario se comunique con la aplicación. Los servicios son componentes de aplicación que ejecutan tareas en background.

Los proveedores de contenido son componentes que permiten compartir datos entre diferentes aplicaciones Android. Los componentes Broadcast Receivers reciben mensajes enviados por el sistema o por otras aplicaciones.

Los componentes que integran una aplicación son especificados en el archivo manifiesto de la aplicación Manifest.xml[?], donde adicionalmente se declaran: tanto los permisos requeridos por la aplicación para acceder a partes protegidas de la API[?] e interactuar con otras aplicaciones; como los permisos requeridos por aplicaciones externas para interactuar con los componentes de la aplicación.

La comunicación entre varias aplicaciones Android(Comunicación interApp), tiene lugar a través de intents[?], métodos proveídos por la API Android para la activación de componentes tanto al interior de la aplicación como entre aplicaciones externas.

II-B. Técnicas de análisis de código

II-B1. Análisis estático y dinámico: Las soluciones propuestas para detectar fuga de información en aplicaciones Android, se enmarcan en el análisis estático o dinámico de la aplicación, en algunos casos, se combinan ambos tipos. En **análisis estático**[2], se estudia el código del programa para inferir todos los posibles caminos de ejecución. Esto se logra construyendo modelos de estado del programa, y determinando los estados posibles a alcanzar por el programa. No obstante, debido a que existen múltiples posibilidades de ejecución, se opta por construir un modelo abstracto de los estados del programa. La consecuencia de tener un modelo aproximado es pérdida de información y posibilidad de menor precisión en el análisis.

Por otro lado, en **análisis dinámico** se ejecuta el programa y se analiza su comportamiento, verificando el camino de ejecución que ha tomado el programa. Esa exactitud en la ejecución que se verifica da precisión al análisis, porque no es necesario construir un modelo aproximado de todos los posibles caminos de ejecución.

II-B2. Técnicas utilizadas en análisis estático: Generalmente, para verificar el cumplimiento de políticas de seguridad mediante análisis estático, se aplican técnicas de seguridad de tipado (Typed-Inference/Security-Typed Analysis) y técnicas de flujo de datos(Data/Control Flow Analysis)[3].

Con **técnicas Security-Typed** las propiedades de confidencialidad e integridad son anotadas en el código, y verificadas a tiempo de compilación, garantizando su cumplimiento a tiempo de ejecución.

Con **técnicas de flujo de control y técnicas de flujo de datos**, las políticas de seguridad son verificadas haciendo seguimiento al control de flujo, o al flujo de datos, respectivamente. Estas técnicas suelen utilizar grafos de Control de Flujo CFG(Control Flow Graph), Grafos de Flujo de Datos DFG(Data Flow Graph) y Grafos de llamadas CG (Call Graphs).

II-B3. Security Typed Languages: Las herramientas basadas en técnicas de análisis Security-Typed, involucran conceptos como flujo de información, políticas de confidencialidad e integridad, y chequeo de tipos. *Flujo de información:* el flujo de información describe el comportamiento de un programa, desde la entrada de los datos hasta la salida de los mismos.

Políticas de confidencialidad e integridad: confidencialidad e integridad son políticas de seguridad aplicables mediante control de flujo de información. Mientras la confidencialidad busca prevenir que la información fluya hacia destinos no apropiados, la integridad busca prevenir que la información provenga de fuentes no apropiadas[4].

Chequeo de tipos: al usar un lenguaje tipado de seguridad, las políticas son definidas a través del lenguaje, porque son expresadas mediante anotaciones en el código fuente del programa a verificar, y su evaluación se realiza mediante chequeo de tipos.

El chequeo de tipos consiste en una técnica estática, también utilizada para analizar flujo de información durante la compilación de un programa, más específicamente en la etapa de análisis semántico, el compilador identifica el tipo para cada expresión del programa y verifica que corresponda al contexto de la expresión. Bajo este principio de chequeo, lenguajes tipados de seguridad aplican políticas de control de flujo, definiendo para cada expresión del programa un tipo de seguridad(security type), de la forma: tipo de dato y label de seguridad(security label). Donde el label de seguridad regula el uso del dato, acorde a su tipo.

El compilador realiza el chequeo de tipos, partiendo del conjunto de labels de seguridad. Así, si el programa pasa el chequeo de tipos y compila correctamente, se espera que cumpla con las políticas de control de flujo evaluadas.

II-C. JIF(Java Information Flow)

Jif es un lenguaje tipado de seguridad que extiende al lenguaje Java con labels de seguridad, a través de los cuales se especifican restricciones de cómo debería ser utilizada la información. Jif está compuesto por un compilador y un sistema de anotaciones.

El análisis de flujo de información de aplicativos Java mediante Jif, requiere su implementación haciendo uso del sistema de

anotaciones de Jif, de modo que se especifiquen las políticas de seguridad a evaluar. Tal implementación se basa en adicionar labels de seguridad a la definición de métodos, variables, arrays, etc; los labels de seguridad no especificados son generados automáticamente con labels por defecto.

La verificación del cumplimiento de las políticas de seguridad, tiene lugar durante la compilación del aplicativo, allí el compilador Jif aplica chequeo de labels(label checking)[5], verificando que los flujos de información generados cumplen con las restricciones establecidas.

II-C1. DML(Decentralized Label Model): Jif basa su sistema de anotaciones en el modelo de etiquetas DLM, donde se manejan tres elementos fundamentales: Principales, Políticas y Etiquetas.

Principales: un principal es una entidad con autoridad para observar y cambiar aspectos del sistema. Un programa pertenece a un principal, quien determina el comportamiento que este debería tener. Jif cuenta con una serie de principals ya definidos, por ejemplo, Alice, Bob, Chunck, etc, que pueden ser utilizados al momento de anotar.

Políticas: mediante políticas de seguridad el dueño de la política, que es el principal que la define, determina qué otros principals pueden leer o influenciar la información. Así, una política puede ser de confidencialidad o de integridad, y se especifican de la forma: {owner: reader list} u {owner: writer list}.

Etiquetas: una etiqueta consiste en un conjunto de políticas de confidencialidad e integridad. Las etiquetas se escriben en las expresiones del programa que se anota(etiquetas de seguridad), esto es métodos, variables, arrays, etc.

En síntesis, las políticas de seguridad definen que Principales pueden leer o modificar la información, y esas políticas se expresan mediante etiquetas.

II-C2. Chequeo de etiquetas: Para hacer seguimiento al flujo de información de un programa, el compilador de Jif asocia una etiqueta al program counter de cada punto del programa, etiqueta del program-counter(pc). En cada punto del programa, el (pc) representa la información que podría conocerse tras la ejecución de ese punto del programa. El (pc) es afectado por las etiquetas con que se define cada sentencia y expresión del programa, por tanto este es considerado como el límite superior(máxima información que podría conocerse) de las etiquetas que han afectado el flujo de información para llegar a un determinado punto de ejecución.

II-C3. Anotación de variables y métodos en Jif: dado que la sintaxis de anotación de Jif se basa en etiquetas de seguridad para extender la sintaxis del lenguaje java, a continuación se ilustra la forma en que normalmente se declaran variables y métodos en Java, y la forma en que se realiza la respectiva extensión con la sintaxis de Jif.

Definición de variables

En Java la sintaxis para definir una variable es:

```
modifier java-type varName
```

Extendiendo la sintaxis Java, en Jif las variables se definen de la forma:

```
modifier java-type {L} varName
```

Donde java-type especifica el tipo de dato Java que almacena la variable, {L} la etiqueta de seguridad para especificar quien es el principal dueño de la variable, y varName, el respectivo nombre de la variable.

Definición de métodos

en Java la definición de un método tiene la siguiente sintaxis:

```
modifier java-type methodName(java-type arg1,...,
java-type argn){body method}
```

En Jif es posible asociar una etiqueta de seguridad al tipo de dato retornado, los argumentos que recibe y las excepciones declaradas. Adicionalmente, se declara un begin-label(BL) y un end-label(EL).

Cuando en la definición del método no se especifica una etiqueta de seguridad, el compilador de Jif asume unas por defecto. La sintaxis es la siguiente:

```
modifier java-type{RTL} methodName{BL}(java-type arg1{AL},...,
java-type argn{AL}) :{EL}
```

Donde: *java-type*, es el tipo de dato Java retornado por el método.

RTL, Return Type Label, indica la etiqueta de seguridad para el valor retornado por el método.

BL, Begin Label, representa el máximo nivel de seguridad del pc desde donde se invoca el método, de este modo, la etiqueta del program counter desde donde se invoca el método debe ser menor o igual de restrictivo que el BL con que se define el método. El BL también asegura que el método sólo podrá actualizar partes del programa que tengan igual BL. Con tales restricciones se evita la generación de flujos implícitos, vía invocación del método.

AL, Argument Label, indica el máximo nivel de seguridad para los argumentos con que se llama el método, así, las etiquetas de los argumentos con que se invoca el método deben ser menor o igual de restrictivos que el AL con que ha definido el método.

EL, End Label, indica el pc en el punto de terminación del método, y representa el máximo nivel de información que puede conocerse tras la finalización del método.

II-C4. Labels de anotación que Jif asume por defecto:

Cuando en la declaración de variables o métodos no se especifica su respectivo label de seguridad, Jif lo infiere o genera automáticamente. De acuerdo al tipo de sentencia. Así:

- Variables locales: Jif infiere sus labels, de modo que se respeten las restricciones sobre el flujo de información.

- Arrays: por defecto, Jif define el label público para el Base Label(BL) y el Size Label(SL) de un array.

- Class fields: el label por defecto es { }, que representa información con el menor nivel de confidencialidad. Es el label menos confiable, con este se asegura que información altamente confidencial no podrá ser almacenado en el campo de la clase.

- Métodos: los labels que Jif genera por defecto para la definición de métodos son:

Argument Label(AL): el label por defecto es el Top principal,

es decir que sólo la máxima autoridad podrá leer la información del argumento.

Begin Label(BL): su label por defecto es el Top principal.

End Label(EL): Jif hace un Join de los labels con que se definen las excepciones del método, si el método no tiene excepciones, el label por defecto es el menos restrictivo { }

Return Type Label(RTL): Jif hace un Join de los AL y el EL. Labels para excepciones: el valor del EL.

II-C5. Flujos implícitos y pc: Los flujos implícitos son canales creados durante el control de flujo del programa. Buscando prevenir la fuga de información a través de estos canales, Jif asocia un pc a cada statement y expresión del programa, representando la información que debería conocerse tras su evaluación.

El sistema de tipos de Jif asegura que el pc debe ser por lo menos tan restrictivo como los labels de las variables de que depende el program counter de la sentencia.

En el siguiente ejemplo se ilustra la generación de flujos implícitos:

```
boolean {Alice;} secreto;  
boolean {} publico;  
secreto = true;  
if( secreto )  
    publico = 0;  
else  
    publico = 1;           //Implicit Flow
```

El flujo implícito tiene lugar en el condicional porque la variable *publico*, cuyo nivel de seguridad es bajo (pc={}) permite conocer información de la variable *secreto*, con nivel de seguridad alto (pc={Alice:})

III. DESCRIPCIÓN DEL PROBLEMA

En Android, por defecto, el desarrollador no cuenta con mecanismos para definir políticas de confidencialidad e integridad que regulen el flujo de información de sus aplicaciones. Siendo complejo prevenir fugas de información del usuario, puesto que, el desarrollador carece de herramientas que le garanticen la ausencia de flujos indeseados.

Precisamente, una de las principales preocupaciones de seguridad en aplicativos Android, es la manipulación de información del usuario. Así lo evidencia un estudio reciente de seguridad en dispositivos móviles, publicado por McAfee[1], este señala que una importante cantidad de aplicaciones Android invaden la privacidad del usuario, reuniendo información detallada de su desplazamiento, acciones en el dispositivo, y su vida personal. De este modo, 80 % reúnen información de la ubicación, 82 % hacen seguimiento de alguna acción en el dispositivo, 57 % registran la forma de uso del celular (mediante Wi-Fi o mediante la red de telefonía), y 36 % conocen información de las cuentas de usuario.

Las motivaciones para este tipo de acciones varían acorde al tipo de información, por ejemplo: monitorear información de ubicación para mostrar publicidad no solicitada; seguir las acciones sobre el dispositivo, para conocer qué aplicaciones son rentables de desarrollar, o para ayudar a aplicaciones maliciosas a evadir defensas; acceder a información de cuentas

del usuario con fines delictivos; obtener información de contactos y calendario del usuario, buscando modificar los datos; obtener información del celular (número, estado, registro de MMS y SMS) para interceptar llamadas y enviar mensajes sin consentimiento del usuario.

Con o sin autorización de acceso, existen motivaciones suficientes para que un tercero desee manipular información del usuario.

Adicionalmente, el informe señala que una aplicación invasiva no necesariamente contiene malware, y que su finalidad no siempre implica fraude; de las aplicaciones que más vulneran la privacidad del usuario, 35 % contienen malware.

Si bien, aplicaciones invasivas no necesariamente implican malware y/o acciones delictivas, el cuestionamiento de fondo es la forma y finalidad con que una aplicación manipula la información del usuario, y qué garantías puede ofrecer el desarrollador para que tal manipulación sea consentida.

La falta de control sobre los flujos de información de la aplicación puede ocasionar fugas de información, generando problemas de seguridad tanto para quien la implementa como para quien la usa.

Como contramedida a este problema, la API de Android ofrece herramientas de seguridad basadas en políticas de control de acceso, y el desarrollador puede implementarlas en su aplicación. Sin embargo, estos mecanismos se centran en regular el acceso de los usuarios del sistema a determinados recursos, y no en verificar qué sucede con la información una vez es accedida.

Para superar dicha carencia, diferentes trabajos de investigación han abordado el problema de fuga de información en aplicaciones Android, tanto desde un enfoque dinámico como desde un enfoque estático, la literatura existente al respecto (TaintDroid[6], Flow-Droid[7], DidFail[8], DroidForce[9]), indica que la mayoría de propuestas hacen data-flow analysis mediante técnicas de análisis tainting, partiendo del bytecode Enfocándose en la precisión y eficiencia del análisis para detectar fugas de datos en aplicaciones de terceros ya implementadas. Por consiguiente, la finalidad del análisis no es garantizar el cumplimiento de políticas de confidencialidad e integridad desde la construcción del aplicativo.

Partir de tales propuestas para analizar aplicaciones propias y garantizar políticas de confidencialidad e integridad desde su construcción, puede implicar incompletitud en el análisis (under-tainting) y no detección de flujos implícitos. Esto debido a que, por un lado, al realizar análisis tainting de forma dinámica, el marcado de datos se propaga únicamente a través de caminos del programa actualmente ejecutados. Así, si existen datos que son influenciados por los datos marcados, pero no están dentro de los actuales caminos de ejecución, quedan sin la propagación de la marca, dando lugar al problema de undertainting[10][11]. Es decir, se obtiene precisión en el análisis, pero se pierde completitud.

Por el otro, aún cuando se hace análisis tainting de forma estática, y el marcado de datos puede ser propagado para todos los caminos posibles de ejecución del programa, superando el inconveniente de under-tainting, la detección de flujos implíci-

tos es posible si, en la construcción de la herramienta de análisis se propaga el marcado de datos para flujos implícitos[12]. Sin embargo, las propuestas que basan su análisis en data-flow estático, suelen restringir la propagación del marcado de datos a flujos explícitos, ganando eficiencia en el análisis. Las propuestas mencionadas anteriormente, no son ajenas a tal generalidad(DidFail[8][page 33], FlowDroid[7][page 30]).

Ahora bien, la falta de garantías en el cumplimiento de determinadas políticas de seguridad en la aplicación que se implementa, puede superarse usando control de flujo de información, Information Flow Control(IFC), puesto que, con esta técnica la aplicación es analizada estáticamente para identificar todos los posibles caminos que podrían tomar sus flujos de información, garantizando que a tiempo de ejecución, la aplicación respeta determinadas políticas de seguridad.

Finalmente, partiendo del contexto que se plantea, donde es el propio desarrollador Android quien requiere evaluar políticas de seguridad en su aplicación, para garantizarle al usuario que la aplicación las cumple. Resulta apropiado proveerle una herramienta de apoyo, mediante la cual analice el flujo de información de la aplicación que implementa, y verifique el cumplimiento de políticas de seguridad.

IV. TRABAJOS RELACIONADOS

Dentro de las propuestas existentes están

V. PROPUESTA

Para garantizarle al usuario que la aplicación que implementa respeta determinadas políticas de seguridad, el desarrollador Android requiere de una herramienta que le permita: primero definir las políticas de seguridad a evaluar, y segundo, verificar el cumplimiento de las políticas definidas.

Así pues, la propuesta para cumplir con tales requerimientos consiste en proveer una herramienta de análisis de flujo de información, mediante en el sistema de anotaciones del lenguaje tipado de seguridad Jif. Se propone partir de Jif porque este ofrece un sistema de anotaciones basado en un modelo de etiquetas(DLM), más un módulo de verificación(compilador de Jif), tal como se ilustra en la figura 1. De este modo, con anotaciones en el código de su aplicación, el desarrollador puede definir políticas de seguridad, para luego validar con el módulo de verificación, si la aplicación respeta tales políticas. Ahora bien, el principal reto para llevar a cabo dicha propuesta, es permitir el análisis de flujo de información de aplicativos Android con el sistema de anotaciones de Jif, puesto que, Jif sólo permite el análisis de flujo de información en aplicativos Java convencionales, y una de las diferencias entre una aplicación Android y una aplicación Java convencional es la API de Android, puesto que, para adquirir sus funcionalidades una aplicación Android requiere de clases de la API. No obstante, el sistema de anotaciones de Jif está implementado para clases del lenguaje Java estandar y no para clases de la API Android. En consecuencia, el diseño ideal para contribuir con la solución del problema es: una herramienta que contenga el setup de Jif para Android, e integre un clasificador de sources y sinks. De modo que, la herramienta analice flujos de información en

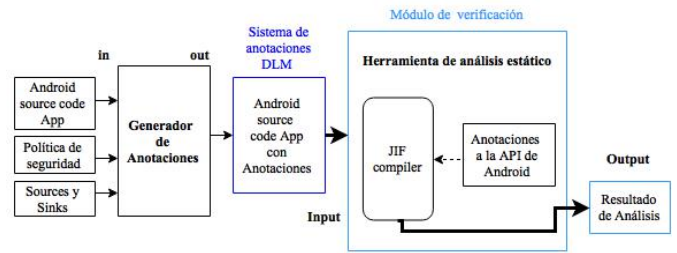


Figura 1: Diseño herramienta de análisis estático.

Partiendo del código fuente del aplicativo Android, la política de seguridad a evaluar, más los sources y sinks requeridos para verificar la política; el generador de anotaciones retorna la versión anotada del aplicativo a analizar. Luego la versión anotada del aplicativo se pasa como entrada a la herramienta de análisis estático, la cual retorna el resultado de análisis de flujo de información.

aplicativos Android, verificando que cumplan con las políticas de seguridad que el desarrollador ha definido.

El Setup de Jif para Android hace referencia a las clases de la API Android con anotaciones Jif(Anotaciones a la API), estas anotaciones son necesarias porque es a través de la API que la aplicación escribe o lee información de los sources y los sinks, generando flujos de información entre los mismos. Por ende, el análisis de flujo de información entre sources y sinks con el sistema de anotaciones de Jif, implica anotaciones a clases de la API.

Si bien, el diseño ideal requiere anotaciones a toda la API, para efectos del presente trabajo se limita el Setup de Jif, partiendo de una política de seguridad específica. Por consiguiente, el diseño se centra en soportar un conjunto reducido de clases de la API de Android, y en incluir un conjunto específico de sources y sinks; de acuerdo a una política de seguridad establecida. Ese conjunto de sources y sinks, se toma del listado de sources y sinks proveído por SuSi ??.

Adicionalmente, para aspectos de evaluación, se incluye el diseño de un anotador que automatiza la anotación requerida por el desarrollador. De modo que, acorde a la política de seguridad establecida, se genere la versión anotada del aplicativo a analizar.

La figura 1 muestra el esquema del diseño, allí los componentes principales son el *generador de anotaciones* y la *herramienta de análisis estático*.

Para retornar la versión Jif del aplicativo, el *generador de anotaciones* parte del código fuente de la aplicación Android a analizar, la política de seguridad a evaluar, y los sources y sinks requeridos para verificar tal política.

Luego esa versión Jif del aplicativo se debe pasar como entrada a la herramienta de análisis estático, la cual retorna el análisis de flujo de información.

La *herramienta de análisis estático*, está integrada por el compilador de Jif(Módulo de verificación) y las anotaciones a la API de Android.

Siguiendo el esquema de diseño anteriormente descrito: primero, se define la política de seguridad a evaluar ??; segundo, se toman a consideración elementos influyentes para verificar el cumplimiento de la política mediante Jif ??; y tercero, teniendo en cuenta ?? y ??, se definen los lineamientos de anotación ?. Tales lineamientos establecen el esquema para anotaciones a la API Android y anotaciones a los aplicativos a analizar(lineamientos del anotador).

V-A. Definición de la política de seguridad

Detectar si una aplicación Android presenta flujos de información entre: información con nivel de seguridad alto e información con nivel de seguridad bajo.

Información con nivel de seguridad alto: la información catalogada con nivel de seguridad alto hace referencia a un conjunto de sources. Los sources representan información confidencial o privada del usuario, por tanto, el usuario quisiera tener el control de hacia donde se dirige tal información. El conjunto de sources está integrado por los métodos `getDeviceId`, `getSimSerialNumber`, `findViewById`, `getLatitude`, `getLongitude` y `getSubscriberId`. Adicional a estos métodos, se incluye el tipo de dato `EditText`, si y sólo si, el campo UI al que hace referencia corresponde a un campo tipo `textPassword`(campos destinados a almacenar contraseñas).

Información con nivel de seguridad bajo: la información considerada con nivel de seguridad bajo, comprende un conjunto de sinks. Los sinks son canales que permiten la salida de información del dispositivo, por ejemplo, los mensajes de texto. El conjunto de sinks está integrado por los métodos de la clases `Log` y `SmsManager` de la API.

Este conjunto de sources y sinks es tomado del listado proveído por el clasificador SuSi ?.

V-B. Lineamientos de anotación

Los lineamientos de anotación definen los elementos básicos de anotación V-B1, anotaciones necesarias para la API de Android V-B2 y anotaciones en los aplicativos a analizar V-B3.

V-B1. Elementos básicos de anotación: Para anotar la información con su respectivo nivel de seguridad alto o bajo, de modo que, partiendo de tales anotaciones se evalúe la existencia de flujos de información entre información con nivel de seguridad alto e información con nivel de seguridad bajo, se define una autoridad para los programas, también se definen las etiquetas de seguridad para especificar las políticas de seguridad. Así:

Principal Alice: haciendo uso de los principales ya definidos en Jif, se establece al principal *Alice* como la autoridad máxima. Este principal tendrá todo el poder para actuar sobre aspectos de los programas.

Política para anotar información con nivel de seguridad alto: la etiqueta de seguridad `{Alice:}`, indica que la información tiene nivel seguridad alto, es decir, que se trata de información sensible o privada.

Variables con nivel de seguridad alto deben ser anotadas con tal label de seguridad, porque esté específica que sólo el dueño de la información(Alice) puede acceder a la misma.

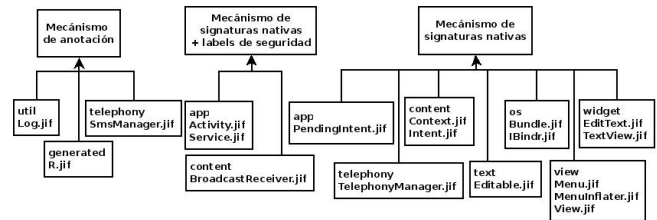


Figura 2: Mecanismos de anotación para clases de la API.

Política para anotar información con nivel de seguridad bajo: la etiqueta de seguridad `{}`, indica que la información tiene nivel de seguridad bajo, es decir, información de conocimiento público.

V-B2. Anotaciones a la API de Android: dado que la API de Android provee las librerías necesarias para implementar las funcionalidades de una aplicación Android, permitiendo a su vez, que el aplicativo escriba o lea información de los métodos sources y sinks, las clases pertenecientes a la API generan flujos de información. Así, para verificar la política de seguridad previamente definida, se requieren anotaciones tanto a las clases que definen los sources y sinks en que se centra la política, como anotaciones a clases y librerías requeridas en la implementación de las aplicaciones, por ejemplo, las clases que permiten implementar componentes de actividades, servicios y broadcast receivers.

En el caso de los sinks la anotación tiene un propósito adicional, este es, controlar el flujo de información que se envía a través de los mismos, la definición de los métodos de las clases `Log` y `SmsManager` de la API Android, deben anotarse de tal manera que, se controle el nivel de seguridad de los argumentos con que se invocan tales métodos.

Para esto se utilizan las etiquetas de seguridad que regulan el llamado a métodos en el sistema de anotaciones de Jif(sintaxis de anotación II-C3), estas son `Begin Label(BL)` y `Argument Label(AL)`. Con las etiquetas para el BL se determinan los puntos del programa desde donde puede invocarse el método, de este modo, el método podrá ser invocado si: el nivel de seguridad del punto del programa desde donde se llama el método es menor o igual de restrictivo que el BL con que ha sido definido el método. Para el presente caso se busca que un método pueda ser invocado desde cualquier punto del programa.

Con las etiquetas para los argumentos del método, AL, se controla el nivel de seguridad de los argumentos con que se invoca el método. Por tanto, el nivel de seguridad de los argumentos con que se invoca un método, debe ser menor o igual de restrictivo que el nivel de seguridad con que han sido definidos los argumentos del método.

El nivel de restricción de la información se evalúa de acuerdo a las etiquetas de seguridad con que se defina e invoque el método. De este modo, información con nivel de seguridad alto es más restrictiva que información con nivel de seguridad bajo. Aterrizando estos conceptos a los elementos definidos

en la sección ??, información anotada con el label `{Alice:}` es más restrictiva que información anotada con el label `{}`. Puesto que, el primer label denota que la información tiene nivel de seguridad alto, mientras que el segundo, indica que la información tiene nivel de seguridad bajo.

Tomando como ejemplo el método `sendMessage` de la clase `SmsManager`, mediante el cual se envían mensajes de texto:

```
sendMessage(String destinationAddress, String scAddress,
            String text, PendingIntent sentIntent,
            PendingIntent deliveryIntent){}
```

Se tiene que el parámetro `text` es el que recibe la información a enviar, y por tanto esa información debe ser pública.

Por consiguiente, en la definición del método la etiqueta de seguridad del argumento(AL) correspondiente a la información a enviar(sms), se anota con la etiqueta pública `{}`. Con tal etiqueta se garantiza que la información se envía, si y sólo si, el nivel de seguridad del argumento con que se invoca el método es menor o igual de restrictivo que la etiqueta pública `{}`; como no se tiene una etiqueta de seguridad menos restrictiva que esta, se garantiza que la información sólo es enviada si efectivamente está anotada con tal etiqueta, es decir, si se trata de información pública.

Por ejemplo, si el método se invoca con información anotada con label de seguridad alto, se genera un flujo de información indebido, dando lugar a un error en la compilación del programa que le llama.

En el caso del BL al anotarlo con la etiqueta `{Alice:}`, se permite que el método sea invocado desde cualquier punto de un programa, es decir, desde puntos con nivel de seguridad bajo(`{}`) y desde puntos con nivel de seguridad alto(`{Alice:}`). Para el resto de labels se dejan los que Jif genera por defecto II-C4.

Continuando con el ejemplo del método `sendMessage` y aplicando lo anteriormente descrito, este método se anota de la siguiente manera:

```
sendMessage{Alice:}(String{Alice:} destinationAddress,
                    String{Alice:} scAddress,
                    String{} text,
                    PendingIntent{Alice:} sentIntent,
                    PendingIntent{Alice:} deliveryIntent) {}
```

El principio de anotación propuesto es aplicable para los métodos de la clase `Log`, ya que estos también representan sinks.

Tales criterios de anotación son aplicados a través del mecanismo fundamental de anotación que provee el sistema de anotaciones de Jif, en el cual se implementa la respectiva versión Jif de la clase, esto es, anotando con etiquetas de seguridad las variables, definición y cuerpo de los métodos de la clase. Adicional a este mecanismo de anotación(mecanismo de anotación¹), el compilador de Jif provee un mecanismo que permite interactuar con código de clases Java ya existentes[13], para esto, se recurre a firmas nativas. Así, se implementa

¹Para referenciar los mecanismos utilizados en la integración de clases de la API Android, al sistema de anotaciones de Jif, se adoptan los nombres: mecanismo de anotación, mecanismo de firmas nativas y mecanismo de firmas nativas más labels de seguridad.

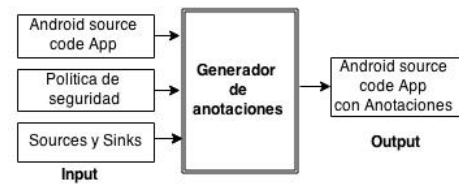


Figura 3: Entradas y salidas para el generador de anotaciones. Para generar la versión anotada del aplicativo a analizar, el anotador parte del código fuente del aplicativo, la política de seguridad definida en ?? y el conjunto de sources y sinks implicados en la misma.

una versión Jif de una clase Java ya existente, en la que se declaran firmas nativas proveídas por Jif, constructor y métodos necesarios de la clase(mecanismo de firmas nativas). Al mecanismo de firmas nativas también se puede adicionar labels de seguridad(mecanismo de firmas nativas más labels de seguridad).

La figura 2 resume las clases a anotar de acuerdo al mecanismo con que se integran al sistema de anotaciones de Jif.

V-B3. Anotaciones en los aplicativos a analizar: Si bien, el desarrollador debe implementar manualmente las anotaciones en el aplicativo a analizar, acorde a las políticas de seguridad que requiere verificar; en el presente diseño de solución, se incluye un generador de anotaciones que automatiza la anotación requerida por el desarrollador. Como se ilustra en la figura 3, el generador de anotaciones retorna la versión anotada del aplicativo a analizar, partiendo del código fuente del aplicativo, la política de seguridad y el conjunto de sources y sinks, necesarios para verificar la política de seguridad.

La anotación del aplicativo se fundamenta en identificar la declaración de sources, y en verificar qué métodos influyen esa información, de modo que, cuando la información sea enviada a través de sinks, tenga el nivel de seguridad adecuado. Así, las políticas de anotación definidas en V-B1 son aplicadas a variables y métodos, de acuerdo a si contienen o no información del conjunto de sources: el tipo de dato `EditText`² y los métodos: `getDeviceId`, `getSimSerialNumber`, `findViewById`, `getLatitude`, `getLongitude` y `getSubscriberId`. Finalmente, se define una serie de pasos que permiten aplicar las políticas de anotación definidas, y esos pasos son automatizados mediante el generador de anotaciones.

VI. EVALUACIÓN

VI-A. Conjunto de evaluación y métricas

Para la evaluación se parte de DroidBench versión 1.0[14], DroidBench es un benchmark creado específicamente para evaluar propiedades de seguridad en aplicaciones Android. De este benchmark se toma un grupo de testcases que permiten evaluar la política de seguridad establecida ??, tal grupo está integrado por 20 testcases, de los cuales, 14 presentan fugas

²Este tipo de dato es considerado como source si y sólo si, el campo UI al que referencia corresponde a un campo tipo `textPassword`, es decir, un campo que almacena contraseñas.

Conjunto de Evaluación	
Test	Nombre Testcase
1	AndroidSpecific_DirectLeak1
2	AndroidSpecific_InactiveActivity
3	AndroidSpecific_LogNoLeak
4	AndroidSpecific_Obfuscation1
5	AndroidSpecific_PrivateDataLeak2
6	ArraysAndLists_ArrayAccess1
7	ArraysAndLists_ArrayAccess2
8	GeneralJava_Exceptions1
9	GeneralJava_Exceptions2
10	GeneralJava_Exceptions3
11	GeneralJava_Exceptions4
12	GeneralJava_Loop1
13	GeneralJava_Loop2
14	GeneralJava_UnreachableCode
15	ImplicitFlows_ImplicitFlow1
16	ImplicitFlows_ImplicitFlow2
17	ImplicitFlows_ImplicitFlow4
18	Lifecycle_ActivityLifecycle3
19	Lifecycle_BroadcastReceiverLifecycle1
20	Lifecycle_ServiceLifecycle1

Tabla I: Conjunto de evaluación.

Donde *Item* identifica el testcase y *Testcase* especifica el nombre de la aplicación para el caso de prueba

de información. En la tabla I se listan los casos de prueba. Este conjunto de prueba es analizado con FlowDroid, JoDroid y con el Prototipo. El Prototipo representa la herramienta de análisis propuesta, por tanto, con el Prototipo se genera la versión anotada del aplicativo y se evalúa la política anotada. Los calificativos para los resultados del análisis son: True Positive(TP): cuando se reporta un leak que efectivamente existe. False Positive(FP): cuando se reporta un leak que no existe. True Negative(TN): cuando no se reporta leak y efectivamente no existe. False Negative(FN): cuando no se reporta un leak existente.

Adicionalmente, con el comando *time*[15] de unix, se mide el tiempo(desempeño) que tarda cada herramienta en ejecutar el análisis.

En base a los resultados de análisis que reporta cada herramienta, se calcula su respectiva Precisión y Recall.

La **Precisión** hace referencia a los Casos Positivos esperados(correctos e incorrectos: TP, FP), en contraste con la proporción de verdaderos Positivos(TP) detectados[16]. Una alta Precisión indica que la herramienta reporta más correctos Positivos(TP) que incorrectos Positivos(FP).

El **Recall** indica la proporción de Casos Positivos detectados(TP), frente a los Casos Positivos esperados como correctos[16]. Un alto Recall indica que la herramienta reporta más correctos Positivos(TP) que incorrectos Negativos(FN), es decir, la herramienta deja pasar menos errores.

Las fórmulas para calcular Precisión(p) y Recall(r), son:

$$p = TP / (TP + FP) \quad (1)$$

$$r = TP / (TP + FN) \quad (2)$$

Resultados de evaluación							
Test	Leaks	F	J	P	tF	tJ	tP
1	1	TP	TP	TP	5.371s	22m11.991s	2.063s
2	0	TN	FP	FP	3.255s	22m25.617s	2.469s
3	0	TN	TN	TN	5.505s	21m6.548s	2.946s
4	1	TP	TP	TP	6.734s	22m46.541	2.706s
5	1	TP	TP	TP	6.144s	21m32.447s	2.644s
6	0	FP	FP	FP	4.708s	22m01.926s	1.278s
7	0	FP	FP	FP	4.4s	22m11.023s	1.361s
8	1	TP	FN	TP	6.397s	22m52.134s	2.755s
9	1	TP	FN	TP	5.887s	21m4.434s	1.980s
10	0	FP	TN	FP	6.008s	21m37.040s	2.032s
11	1	TP	FN	TP	5.731s	21m10.240s	2.313s
12	1	TP	TP	TP	5.605s	21m15.30s	2.800s
13	1	TP	TP	TP	4.719s	21m41.224s	1.361s
14	0	TN	TN	FP	3.792s	22m84.138s	1.197s
15	1	FN	TP	TP	4.853s	22m55.645s	1.331s
16	1	FN	TP	TP	4.496s	22m32.231s	1.212s
17	1	FN	TP	TP	4.375s	22m43.110s	1.224s
18	1	TP	TP	TP	4.792s	22m54.651s	1.222s
19	1	TP	TP	TP	4.456s	22m42.347s	1.061s
20	1	TP	TP	TP	5.225s	22m92.722s	1.180s

Tabla II: Resultados de evaluación para FlowDroid, JoDroid y Prototipo. Donde *test* indica el testcase que se evalúa; *Leaks* especifica el numero de fugas de información que presenta el testcase; *F*, *J* y *P* muestran los resultados retornados por FlowDroid, JoDroid y por el Prototipo; *tF*, *tJ* y *tP*, señalan el tiempo que toma el análisis para Flowdroid, JoDroid y para el Prototipo, respectivamente.

VI-B. Comparación de Desempeño: FlowDroid, JoDroid y Prototipo

En la tabla II se ilustran los resultados de analizar el conjunto de pruebas con FlowDroid, JoDroid y el Prototipo. El campo *test* identifica los casos de prueba listados en la tabla I. De acuerdo al tiempo que toma cada herramienta para analizar cada caso de prueba, tal como se registra en los campos *tF*, *tJ* y *tP*; el Prototipo presenta mejor desempeño frente a FlowDroid y JoDroid. En el caso de FlowDroid, la ejecución del análisis, tarda en promedio tarda 3,3 segundos más que el Prototipo. En el caso de JoDroid, el tiempo de análisis es costoso, ya que su tiempo de ejecución oscila entre 21 y 22 minutos.

Como lo señalan los resultados de análisis, el Prototipo presenta mejor desempeño, esto como resultado de analizar flujo de información mediante lenguajes tipados de seguridad, más específicamente a través del lenguaje tipado de seguridad Jif. Dado que Jif recurre a técnicas de compilación y no requiere la generación de grafos de dependencia, el análisis toma menos tiempo.

VI-C. Precisión, Recall y Detección de Flujos Implícitos

Partiendo de los resultados de evaluación de la tabla II, en la tabla III se resume el total de falsos positivos(FP), verdaderos positivos(TP), verdaderos negativos(TN) y falsos negativos(FN), reportados por cada herramienta. Con estos valores se calcula la Precisión y el Recall, cuyos resultados son ilustrados en la tabla IV.

Respecto a los resultados de la tabla IV, es posible decir: **Precisión**: el análisis pesimista en que se basa el Prototipo, donde se asume que todos los métodos implementados en la

	FlowDroid	JoDroid	Prototipo
FP	3	3	5
FN	3	3	0
TP	11	11	14
TN	3	3	1

Tabla III: Resultados de precisión para FlowDroid y Prototipo. Resume el total de FP, TP, TN y FN

	FlowDroid	JoDroid	Prototipo
Precisión	78,57 %	78,57 %	73,68 %
Recall	78,57	78,57 %	100 %
Flujos Implícitos	No	Si	Si

Tabla IV: Comparación entre FlowDroid, JoDroid y Prototipo. Ilustra los porcentajes para Precisión, Recall, y la detección de leaks mediante flujos implícitos.

aplicación serán invocados, hace que los resultados del análisis sean menos precisos, generando más falsos positivos(FP).

Recall: Para la evaluación realizada, el análisis de flujo de información mediante el lenguaje tipado de seguridad Jif, ofrece mejor Recall, frente al análisis de flujo de datos en que se basa FlowDroid y el análisis de flujo de información (mediante System Dependences Graphs) en que se basa JoDroid. En consecuencia, para el experimento de evaluación, la técnica de análisis del Prototipo es completa(completeness), puesto que, dentro de los leaks detectados están todos los leaks que efectivamente existen.

Flujos Implícitos: Una ventaja de las técnicas basadas en control de flujo de información es que al analizar tanto flujos explícitos como flujos implícitos, detectan la generación de leaks mediante flujos implícitos casi de forma natural, contrario a lo que sucede en las técnicas de análisis de flujos de datos, en estas, si la construcción del análisis no define casos de propagación para el marcado de datos mediante flujos implícitos, el análisis carece de criterios para la detección de fugas a través de los mismos.

La tabla V resume las ventajas y desventajas entre el Prototipo y las herramientas de comparación, FlowDroid y JoDroid. En esta se indica si la herramienta presenta mayor(+) o menor(-): precisión, recall y costo en desempeño. Si la herramienta permite(✓) o no permite(X): detección de fugas de información a través de flujos implícitos, detección automática para sources y sinks, y soporte para análisis entre varias aplicaciones.

Finalmente, los resultados de evaluación confirman las hipótesis iniciales del presente trabajo, según las cuales se esperaba que: primero, al hacer análisis de flujo de información mediante lenguajes tipados de seguridad, la herramienta estaría en capacidad de detectar fugas de información tanto en flujos explícitos como en flujos implícitos; segundo, los resultados del análisis serían más rápidos pero menos precisos, reportando más falsos positivos

Item	Prototipo	FlowDroid	JoDroid
Precisión	-	+	+
Recall	+	-	-
Costo en desempeño	-	-	+
Detección de Flujos Implícitos	✓	X	✓
Detección automática de sources y sinks	X	✓	X
Soporte para análisis InterApp	X	✓	X

Tabla V: Síntesis ventajas y desventajas del Prototipo frente a FlowDroid y JoDroid,respectivamente.

que JoDroid y FlowDroid.

VII. TRABAJO FUTURO

VIII. CONCLUSIONES

REFERENCIAS

- [1] McAfee. (2014, February) Who's watching you?, mcafee mobile security report. [Online]. Available: <http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf>
- [2] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *WODA 2003 International Conference on Software Engineering (ICSE) Workshop on Dynamic Analysis*, ser. ICSE'03, Portland, Oregon, 2003, pp. 25–28. [Online]. Available: <http://www.cs.nmsu.edu/~jcook/woda2003/>
- [3] S. Genaim and F. Spoto, "Information flow analysis for java bytecode," *Proceeding VMCAI'05 Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, 2005.
- [4] AndreiSabelfeld and A. wC.Myers, "Language-based information-flow security," *IEEE Journal*, vol. 21, no. 1, pp. 1–15, January 2003.
- [5] Cornell University. (2014, March) Jif reference manual. [Online]. Available: <http://www.cs.cornell.edu/jif/doc/jif-3.3.0/language.html#unsupported-java>
- [6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *9th USENIX Symposium on Operating Systems Design and Implementation(OSDI'10)*, pp. 1–15, October 2010.
- [7] C. Fritz, "Flowdroid: A precise and scalable data flow analysis for android," Master's thesis, Technische Universität Darmstadt, July 2013.
- [8] A. S. Bhosale, "Precise static analysis of taint flow for android application sets," Master's thesis, Heinz College Carnegie Mellon University Pittsburgh, PA 15213, May 2014.
- [9] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "Droidforce: Enforcing complex, data-centric,system-wide policies in android," *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*, pp. 1–10, September 2014.
- [10] C. Hammer and G. Snelting, "Formal characterization of illegal control flow in android system," *Signal-Image Technology Internet-Based Systems (SITIS), 2013 International Conference on*, pp. 293 – 300, Dec. 2013.
- [11] B. Yadegari and S. Debray, "Bit-level taint analysis," *2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014.
- [12] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *ISSTA '07 Proceedings of the 2007 international symposium on Software testing and analysis*, July 2007, pp. 196–206.
- [13] C. University. (2015, Jan) Interacting with java classes. [Online]. Available: <http://www.cs.cornell.edu/jif/doc/jif-3.3.0/misc.html#java-classes>
- [14] Secure Software Engineering Group at EC SPRIDE. (2013, June) Droidbench – benchmarks. [Online]. Available: <https://github.com/secure-software-engineering/DroidBench/blob/e64bf483949bf4cb91af642a415fad9c65e4be5/README.md>
- [15] Panagiotis Christias. (2015, April) Unix on-line man pages. [Online]. Available: <http://dell9.ma.utexas.edu/cgi-bin/man-cgi?00+00>
- [16] D. M. W. Powers, "Evaluation: From precision, recall and f-factor to roc, informedness, markedness correlation," School of Informatics and Engineering Flinders University • Adelaide • Australia, Tech. Rep., 2007.