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—

Controlar el acceso y uso de la información, representa una de las principales preocupaciones de seguridad en Aplicativos Android. 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.

Partiendo de lo anterior, el presente trabajo de investigación plantea aplicar técnicas de análisis basadas en control de flujo de información, con el fin de garantizar el cumplimiento de políticas de seguridad en aplicaciones Android, desde su construcción.

CATEGORÍAS Y DESCRIPCIÓN DE TEMÁTICAS

Análisis de flujos de información en aplicaciones.

TÉRMINOS 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 garantizarle al usuario que la aplicación que le provee no presenta fugas de información.

Ahora bien, aunque 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 pueda verificar el cumplimiento de políticas de seguridad desde la construcción de 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.

Así pues, en el presente trabajo se propone una herramienta para análisis de flujo de información de aplicativos Android mediante el sistema de anotaciones de Jif.

Dado que Jif permite anotar código Java pero no código Android, es decir, las anotaciones Jif son válidas para clases del lenguaje Java estándar, no para clases específicas de la API del framework Android, las cuales son indispensables

para implementar las funcionalidades de aplicativos Android; la principal contribución de la propuesta consiste en: proveerle al desarrollador de aplicativos Android una herramienta que permite definir y verificar políticas de confidencialidad en sus aplicativos, con el sistema de anotaciones de Jif.

El resto del articulo está organizado de la siguiente manera: en la sección II se describe el problema de investigación, en la sección III se presentan conceptos para contextualizar la solución del problema, en la sección IV se plantea la propuesta de solución, en la sección V se presentan las diferencias entre la propuesta planteada y los trabajos relacionados, en la sección VI se presentan los resultados de evaluación de la propuesta, en la sección VII se elaboran algunas discusiones y se plantea el trabajo futuro, finalmente en la sección VIII se presentan las conclusiones.

II. 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[9], Flow-Droid[10], DidFail[11], DroidForce[12]), 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[13][14]. 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 implícitos es posible si, en la construcción de la herramienta de análisis se propaga el marcado de datos para flujos implícitos[15]. 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[11][page 33], FlowDroid[10][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.

III. Contexto

En esta sección se presentan conceptos y herramientas necesarias para contextualizar el tema de investigación: Análisis de flujo de información en aplicaciones Android. Así, la sección inicia describiendo en qué consisten las aplicaciones Android. Luego presenta las técnicas de análisis de código con que suelen analizarse estas aplicaciones. Después, expone los conceptos de sources y sinks. Finalmente, describe un lenguaje tipado de seguridad para análisis de flujo de información en aplicativos Java(Jif).

III-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.
En consecuencia, 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: Actividades, Servicios, Proveedores de contenido 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[2], donde adicionalmente se declaran: tanto los permisos requeridos por la aplicación para acceder a partes protegidas de la API[3] 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[4], métodos proveídos por la API Android para la activación de componentes tanto al interior de la aplicación como entre aplicaciones externas.

III-B. Técnicas de análisis de código

III-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**[5], 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.

III-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)[6].
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. Estás 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).

III-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[7].

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.

III-C. Clasificación de Sources y Sinks(SuSi)

En el ámbito de análisis de seguridad de aplicaciones, independientemente del tipo de análisis, estático o dinámico, el punto de partida es la definición de políticas de seguridad, los pasos sucesivos para detectar la perdida de información giran en torno a tales políticas.
Muchas de las propuestas para análisis de flujo de información en aplicaciones Android, parten de un listado de sources y sinks para definir sus políticas de privacidad. Así, en el grupo de **sources** se incluyen las fuentes de datos sensibles, mientras que en el grupo de **sinks**, se incluyen los medios o canales que podrían filtrar información sensible de forma no autorizada. En consecuencia, la efectividad del análisis se ve limitada al listado de sources y sinks, y la veracidad de los mismos. El inconveniente con estos sources y sinks, es que su clasificación suele hacerse de forma manual, por tanto, existe mayor probabilidad de error u omisión.
Con el fin de precisar dicha clasificación, el trabajo de investigación **SuSi** propone el uso de machine-learning para la clasificación y categorización de sources y sinks, partiendo del código fuente de la API Android. La propuesta de análisis se materializa en una herramienta que: partiendo del código fuente de una determinada versión de la API de Android, retorna un listado con la respectiva categorización de sources y sinks.
El proceso de análisis se compone de dos rondas secuenciales: clasificación y categorización.
Así, la salida de la primera ronda: clasificación de sources y sinks, se convierte en entrada para la ronda de categorización, donde se definen diferentes tipos de categorías, 12 para sources y 15 para sinks.

III-D. 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)[8], verificando que los flujos de información generados cumplen con las restricciones establecidas.

III-D1. 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.

III-D2. 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 progam-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.

III-D3. 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 específica 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 se 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.

III-D4. 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.

III-D5. 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                                    //Implicit Flow
        publico = 1;
```

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:})

IV. Propuesta

La presente sección inicia con una descripción de alto nivel del diseño de solución que se propone. Después, describe detalladamente los elementos que conforman la propuesta de solución: IV-A y IV-B.

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 estándar 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 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(III-C) .
Adicionalmente, para aspectos de evaluación, se incluye el diseño de un anotador que automatiza la anotación requerida por el desarrollador. De manera 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 IV-A; y segundo, se definen los lineamientos de anotación IV-B.

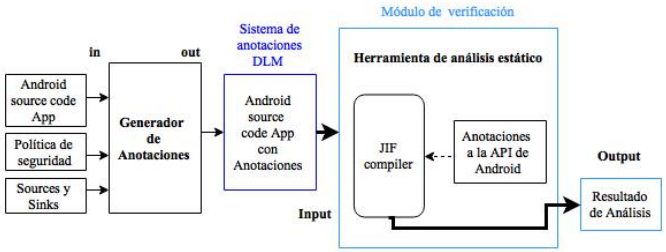

Figura 1: Diseño herramienta de análisis estático.
Partiendo del código fuente del aplicativo Android, la política de seguridad a evaluar, más los sources y sinks requeridos para verificar la política; el generador de anotaciones retorna la versión anotada del aplicativo a analizar. Luego la versión anotada del aplicativo se pasa como entrada a la herramienta de análisis estático, la cual retorna el resultado de análisis de flujo de información.

IV-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.

IV-B. Lineamientos de anotación

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

IV-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

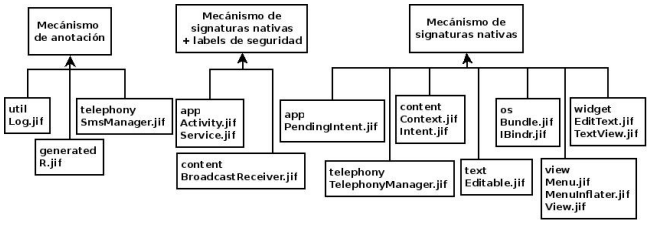

Figura 2: Mecanismos de anotación para clases de la API.

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.
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.

IV-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 III-D3), estas son Begin Label(BL) y Argument Label(AL). Con las etiquetas para el BL se determinan los puntos del programa desde donde puede ser invocado 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 de 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 IV-B1, 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 sendTextMessage de la clase SmsManager, mediante el cual se envían mensajes de texto:

```
sendTextMessage(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(III-D4) .
Continuando con el ejemplo del método sendTextMessage y aplicando lo anteriormente descrito, este método se anota de la siguiente manera:

```
sendTextMessage{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

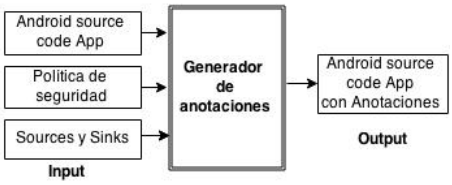

Figura 3: Entradas y salidas para el generador de anotaciones. Para generar la versión anotada del aplicativo a analizar, el anotador parte del código fuente del aplicativo, la política de seguridad definida, más el conjunto de sources y sinks requeridos para la misma.

seguridad las variables, definición y cuerpo de los métodos de la clase. Adicional a este mecanismo de anotación(mecanismo de anotación[1]), el compilador de Jif provee un mecanismo que permite interactuar con código de clases Java ya existentes[16], para esto, se recurre a signaturas nativas. Así, se implementa una versión Jif de una clase Java ya existente, en la que se declaran signaturas nativas proveídas por Jif, constructor y métodos necesarios de la clase(mecanismo de signaturas nativas). Al mecanismo de signaturas nativas también se puede adicionar labels de seguridad(mecanismo de signaturas 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.

IV-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 influencian 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 IV-B1 son aplicadas a variables y métodos, de acuerdo a si contienen o no información del conjunto de sources, integrado por: el tipo de dato EditText(cuando el campo UI al que referencia corresponde a un campo que almacena contraseñas), 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.

[1]Para 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 signaturas nativas y mecanismo de signaturas nativas más labels de seguridad.

V. Trabajos Relacionados

Esta sección presenta otras propuestas de solución existentes para el análisis de seguridad en aplicativos Android. Adicionalmente, indica el elemento diferenciador de la propuesta planteada(IV) frente a estas.

V-A. JOANA

JOANA (Java Object-sensitive ANAlysis)- Information Flow Control Framework for Java[17]. Verifica si una aplicación java contiene fugas de información, mediante análisis estático de flujos de información. El análisis parte de anotaciones en el código fuente de la aplicación. JOANA utiliza técnicas de análisis de flujo de datos y técnicas de análisis de control de flujo. El frontend de la herramienta está basado en el framework de análisis de programas WALA[18], a partir del cual obtiene la representación intermedia del código Java en forma SSA(Static Single Assignement), lo que permite obtener información dinámica del programa. Por otro lado, utiliza Grafos de Dependencia, System Dependence Graphs(SDG), para detectar dependencias entre las sentencias del programa, es decir, si existen caminos entre sentencias etiquetadas con nivel de seguridad alto y sentencias con nivel de seguridad bajo. Para esta etapa del análisis recurre a técnicas de slicing y chopping, reduciendo la cantidad de caminos posibles sólo a los válidos. Así obtiene como resultado, una mayor precisión y reducción de falsas alarmas en el análisis.

Aunque JOANA provee sencillez a la hora de anotar el código a analizar, pues sólo es necesario anotar inputs y outputs del programa, porque la herramienta se encarga de propagar las anotaciones en el resto del programa; carece de características adicionales ofrecidas por sistemas de tipado de seguridad, por ejemplo, el mecanismo downgrading facilitado por JIF.

Si bien, al igual que JOANA, la herramienta propuesta a través del presente trabajo, aplica análisis de control de flujo de información, esta última busca analizar aplicaciones implementadas en código Android, aprovechando las ventajas del sistema de anotaciones de JIF. Proporcionando una herramienta de apoyo al desarrollador de aplicaciones Android, ya que por el momento, JOANA sólo analiza aplicaciones en JAVA.

V-B. JoDroid

JoDroid[19] es una extensión a la herramienta de análisis JOANA para soportar análisis de aplicaciones Android.
El análisis de JOANA está basado en Program Dependence Graphs(PDG) y técnicas slicing. Con PDGs obtiene una representación del programa que analiza, donde los nodos representan statements y expresiones; y las aristas modelan las dependencias sintácticas entre los statements y expresiones: dependencias de datos y dependencias de control, por tanto el grafo está en capacidad de modelar, tanto flujos explícitos como flujos implícitos.
Con técnicas slicing provee sensibilidad al contexto, puesto que el PDG se construye de manera tal que al hacer el backwards slice de un determinado nodo, se obtiene cada

nodo que es alcanzable por caminos del grafo que conservan llamadas al contexto.
El PDG es generado mediante el Front-end de WALA, framework que analiza bytecode de Java. Así, los ajustes hechos a JOANA adaptan parte del Front-end de WALA para generar el PDG de aplicaciones Android.
JoDroid detecta tanto flujos explícitos como flujos implícitos.

V-C. FlowDroid

FlowDroid es una herramienta para análisis estático de flujo de datos en Aplicaciones Android. También permite el análisis de aplicaciones Java.
Esta herramienta utiliza un tipo especial de análisis de flujo de datos: análisis tainting, que hace seguimiento al flujo de datos entre un conjunto de sources y un conjunto de sinks. Define tales conjuntos a partir de SuSi[III-C], un clasificador automático de sources y sinks para la API de Android.
FlowDroid provee un alto recall y precisión[10] en el análisis. El recall, mediante un fiel modelamiento del ciclo de vida de una aplicación Android; la precisión, incluyendo elementos de análisis como: context-, flow-, field- y object-sensitive. Para proveer sensibilidad al flujo y al contexto, recurre a grafos de llamada; y con grafos que modelan todos los procedimientos del programa(inter-procedural control-flow graph), analiza el flujo de datos entre procedimientos, proporcionando field- y object-sensitive.
Los autores de esta propuesta, alcanzan precisión en la construcción del grafo de llamadas extendiendo Soot[20], un framework que genera código intermedio para código Java y código ejecutable Android(dex). Adicionalmente, con el framework Heros[21], incluyen llamadas multihilos en el análisis de flujo de datos entre procedimientos.

Entre las limitaciones de FlowDroid está el over-tainting y la no detección de flujos implícitos. Por tanto, la herramienta no distingue elementos marcados ni dentro de arrays, ni dentro de collections, si se inserta un elemento marcado dentro de alguna de estas estructuras, inmediatamente se marca el resto de elementos. La herramienta tampoco identifica flujos implícitos, puesto que, según los resultados de evaluación de DroidBench[22], su benchmark; cuando Flowdroid analiza el conjunto de aplicaciones de prueba para la identificación de flujos implícitos, no detecta fuga de datos, generando falsos negativos en la detección de flujos implícitos[10, pags 32-36].

Aún cuando el problema a atacar es el mismo: fuga de información, la propuesta que se expone a través del presente trabajo difiere en el enfoque de análisis de FlowDroid, mientras FlowDroid se concentra en detectar si la aplicación de un tercero presenta fugas de información, la herramienta planteada aborda el análisis del lado del desarrollador de la aplicación, apoyándolo en la verificación del cumplimiento de políticas de seguridad. Así, resulta más conveniente guiar el análisis mediante control de flujo de información, ya que se previene fuga por datos no marcados para el análisis(under-tainting) y por la no detección de flujos implícitos, siendo posible garantizar el cumplimiento de políticas de seguridad.

V-D. TaintDroid, Dinamic Taint Tracking, para la detección de fugas de Información

A diferencia de las propuestas expuestas anteriormente, caracterizadas por ejecutar el análisis de manera estática, TaintDroid es una herramienta de análisis dinámico. Está herramienta extiende la plataforma de dispositivos celulares Android, con el fin de verificar el uso dado por aplicaciones de terceros a datos sensibles del usuario. El análisis aplica técnicas de análisis tainting, marcando automáticamente como sources, datos provenientes de fuentes consideradas privadas y/o sensibles; y como sinks, canales para la salida de datos de la aplicación, como por ejemplo internet. Cada vez que un dato marcado como source sale de la aplicación, se genera un log.
Para reducir sobrecarga en el dispositivo, pues el análisis es ejecutado a nivel de instrucciones, instrumentan la máquina virtual de Android con marcas de propagación a nivel de: variables, métodos, mensajes y archivos. Las marcas de variable hacen seguimiento a datos dentro de aplicaciones consideras no confiables. Las marcas de mensaje siguen mensajes entre aplicaciones. Debido a que TaintDroid no hace seguimiento a la ejecución de código nativo, utiliza las marcas de métodos para hacer seguimiento a lo retornado luego de invocar métodos de librerías nativas. Las marcas de archivo son utilizadas para verificar la persistencia de los datos, acorde a las políticas de seguridad.
Otra medida para reducir sobrecarga en la ejecución del análisis, consiste en omitir flujos de control, generando no detección de flujos implícitos[9, pag 12].
Si bien, TaintDroid supera el inconveniente de sobrecarga en la ejecución del análisis, un inconveniente característico en análisis dinámico, está limitado para detectar fuga de datos mediante flujos implícitos, puesto que se enfoca en hacer seguimiento a flujos de datos directos(flujos explícitos).

Al ser una herramienta de análisis dinámico, TaintDroid sólo detecta fugas de información correspondiente a las ejecuciones presentadas por el programa, y para la finalidad de su análisis: informar al usuario de posibles fugas de información, se puede decir que es adecuado. No obstante, para los propósitos de la propuesta planteada a través del presente trabajo, con la que se pretende brindar una herramienta de análisis para que el desarrollador verifique el cumplimiento de políticas de seguridad en la aplicación que implementa, no resulta viable aplicar análisis dinámico, ni técnicas de análisis tainting para hacer seguimiento a flujos de datos.

VI. EVALUACIÓN

Esta sección inicia presentando el conjunto de evaluación del que se parte para evaluar la herramienta de solución propuesta(Prototipo), más las métricas de seguridad a verificar. Después compara los resultados de analizar el conjunto de evaluación con el Prototipo, y otras dos herramientas: FlowDroid y JoDroid.

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 *Test* identifica el caso de prueba y *Testcase* especifica el nombre de la aplicación de caso de prueba.

VI-A. Conjunto de evaluación y métricas

Para la evaluación se parte de DroidBench versión 1.0[22], 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(IV-A), tal grupo está integrado por 20 testcases, de los cuales, 14 presentan fugas 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*[23] 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[24]. 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[24]. Un alto Recall indica que la herramienta reporta más correctos Positivos(TP) que incorrectos Negativos(FN), es decir, la herramienta deja pasar menos errores.

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 cada herramienta: FlowDroid, JoDroid y por el Prototipo; *tF*, *tJ* y *tP*, señalan el tiempo que tarda el análisis con cada herramienta.

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

$$p = TP/(TP + FP) \tag{1}$$

$$r = TP/(TP + FN) \tag{2}$$

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

	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.

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, trae como resultado un análisis menos preciso, ya que, al tratar de incluir todas las posibles ramas de ejecución, se generan más falsos positivos.

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): detectar fugas de información a través de flujos implícitos, detectar automatícenme sources y sinks, y analizar varias aplicaciones(Análisis InterApp).

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 que JoDroid y FlowDroid.

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.

VII. Discusión y Trabajo Futuro

La presente sección incoa con una discusión de las limitaciones y retos que implica la propuesta de solución. Después, plantea el trabajo futuro.

VII-A. Discusión

VII-A1. Límites de la solución propuesta: las limitaciones del diseño de solución en que se enfoca el presente trabajo, corresponden a políticas de seguridad evaluables, tipos de aplicaciones evaluables y limitaciones propias del lenguaje Jif.

Políticas de seguridad evaluables

Se propone un esquema de anotación con niveles de seguridad alto y bajo, que permite definir y evaluar políticas de confidencialidad en aplicativos Android mediante el sistema de anotaciones de Jif. Sin embargo, el esquema de anotación propuesto no permite evaluar políticas de integridad ni aplicar mecanismos Downgrading, características ofrecidas por el sistema de anotaciones de Jif.

Tipos de aplicaciones evaluables

El diseño de solución en que se hace énfasis, para la herramienta de análisis propuesta, no permite hacer análisis de flujo de información vía interApp, es decir, no se hace seguimiento al flujo de información durante el envío de mensajes intents para activar componentes de aplicaciones externas.

Características del lenguaje Jif

El compilador de Jif no soporta algunas características del lenguaje Java estándar como: nested clases, initializer blocks, threads, etc; por tanto no es posible analizar el flujo de información de aplicativos Android que requieran de tales características del lenguaje Java, a menos que se encuentre sintaxis equivalente para soportar tales características.

VII-A2. Retos para el análisis: el análisis de flujo de información en aplicativos Android mediante el sistema de anotaciones de Jif, involucra una serie de retos como consecuencia de: las diferencias entre una aplicación Android y una aplicación Java convencional; y las limitaciones propias del lenguaje Jif.

Por un lado, Jif permite anotar código Java pero no código

Android, es decir, las anotaciones Jif son válidas para clases del lenguaje Java estándar, no para clases específicas de la API del framework Android.
Ahora bien, aunque en esencia una aplicación Android es una aplicación Java con interfaces descritas en sintaxis XML, una aplicación Android requiere de las clases proveídas por el framework para la implementación de funcionalidades.
Así, para analizar aplicativos Android con Jif, se necesita anotar clases de la API Android mediante el sistema de anotaciones de Jif, de modo que se posibilite el análisis de flujo de información.

Por otro lado, la anotación de las clases de la API Android(que en el fondo están implementadas en lenguaje Java), están limitadas a las características del lenguaje Java estándar reconocidas por el compilador de Jif.

Entre los retos que surgen están:

Clases de la API del framework y su soporte para Jif

Anotar las clases de la API Android para el sistema de anotaciones de Jif no es una tarea trivial debido a la cantidad(1) y a características específicas del framework(2).

(1) Cantidad de clases a anotar: para analizar flujo de información en aplicativos Android, lo ideal sería que todas las clases que conforman la API de Android estuviesen anotadas a través del sistema de anotaciones de Jif, no obstante, como la cantidad de clases que conforman la API es bastante amplía, en la presente tesis se anotan únicamente las requeridas para verificar una política de seguridad específica.

(2) Características específicas del framework: entre las funcionalidades del framework está proveer las clases necesarias para interactuar con las interfaces XML. No obstante, las clases involucradas para tales propósitos incluyen operaciones específicas del framework no soportadas mediante el sistema de anotaciones de Jif, puesto que, implican características del lenguaje Java estándar que no son soportadas. Para las cuales, se adoptan mecanismos que permitan analizar la información. Tales características incluyen:
- Casting entre tipos EditText y View: para procesar información proveniente de campos de interfaces XML, se requiere hacer casting entre los tipos de datos representados por las clases EditText y View. Sin embargo, este tipo de casting no es soportado por el compilador Jif.
- Clase Nested R: para referenciar los recursos(strings, estilos, widgets, layouts, e interfaces xml) necesarios para una aplicación, el framework genera la clase R.java. Ahora bien, el inconveniente para anotar ese tipo de clases con Jif, es que los identificadores son descritos mediante clases nested, y esa característica del lenguaje Java está dentro de las limitaciones del lenguaje Jif.
- Sobreescritura de componentes: para la implementación de aplicativos en Android es necesaria la sobreescritura de componentes específicos de la API Android, componentes como (activities, content providers, receivers, services), y cómo para indicar la sobreescritura de los respectivos métodos de un componente se usa el Statement @Override cuya clase no es soportada por el sistema de anotaciones de Jif.

Características específicas del lenguaje Java estándar

Otro reto importante para analizar aplicativos Android a través del sistema de anotación de Jif, es que la sintaxis del lenguaje java en la implementación del aplicativo se restringe a la sintaxis soportada por Jif, por ejemplo: Jif no soporta el enhanced for loop[2]. En consecuencia, habría que extender el compilador de Jif para que efectivamente reconozca características adicionales del lenguaje Java.

En síntesis, los retos emergentes implican extensiones al sistema de anotaciones de Jif, tanto para soportar clases específicas del framework de Android, como para soportar características del lenguaje Java estándar no soportadas por Jif.
Adicionalmente, para las características del framework Android que definitivamente no se puedan soportar mediante el sistema de anotaciones de Jif, es necesario adoptar mecanismos que permitan analizar el flujo de información en las aplicaciones que implementen tales características.

VII-B. Trabajo Futuro

Exceptuando las características del lenguaje Java que no son soportadas por el compilador de Jif(nested clases, initializer blocks, threads, etc), se podría ampliar el setup de Jif para clases de la API de Android, de modo que se anote con el sistema de anotaciones de Jif la mayor cantidad de clases posibles de la API. Esto permitiría hacer análisis de flujo de información a aplicaciones Android más robustas. Por ejemplo, si se anotan todas las clases correspondientes para el manejo de intents(mensajes utilizados principalmente para comunicar diferentes aplicaciones), sería posible incluir en el análisis de flujo de información, el análisis vía interApp.

El esquema de anotación propuesto podría ser extendido para definir y analizar políticas de integridad y mecanismos adicionales como declasificación y endorsement, verificables mediante el modelo de anotaciones de Jif. De este modo, el desarrollador también podría garantizar el cumplimiento de políticas de integridad, y contaría con mecanismos que le permitan flexibilizar la definición de las políticas tanto de confidencialidad como de integridad.

VIII. Conclusiones

El presente trabajo de investigación ha abordado la problemática enfrentada por el desarrollador de aplicaciones Android, a la hora de definir políticas de seguridad que regulen el flujo de información de sus aplicaciones. Puesto que, aún cuando la API Android ofrece mecanismos de control de acceso y el desarrollador puede implementarlos en sus aplicaciones, estos se centran en regular el acceso de los usuarios a determinados recursos del sistema, y no en verificar qué sucede con la información una vez es accedida.

Buscando contribuir con la solución de tal problemática, se propone una herramienta de análisis estático basada en el sistema de anotaciones de Jif, que permita analizar flujo de información en aplicativos Android. El diseño ideal para la

[2]El enhanced for es una versión mejorada del for, utilizado para simplificar la iteración en arrays y colecciones.

propuesta de solución, implica extender el setup de Jif para la API de Android e incluir un clasificador de sources y sinks. Sin embargo, para efectos de la presente tesis se limita el setup y el conjunto de sources y sinks, acorde a una política de seguridad específica.

El diseño de solución en que se hace énfasis para la herramienta de análisis estático, es evaluado y los resultados obtenidos son comparados frente a otras herramientas de análisis estático: FlowDroid y Jodroid. Partiendo de los tipos de análisis y técnicas evaluadas, de sus ventajas y desventajas, se puede concluir:

- Con el sistema de anotaciones de Jif es posible proveer una herramienta de apoyo al desarrollador de aplicaciones Android, de tal manera que evalúe el cumplimiento de políticas de seguridad desde la construcción de sus aplicativos.

No obstante, el desarrollador debe adquirir un conocimiento previo de la implementación de aplicativos en Jif.

- Al estar basado en análisis de flujo de información, Jif analiza tanto flujos explícitos como flujos implícitos, ofreciendo la ventaja de detección de fugas de información a través de flujos implícitos, sin requerirse trabajo adicional para que el análisis incluya tales flujos. Contrario a lo que sucede con las técnicas de análisis tainting, pues para incluir flujos implícitos en el análisis, se requiere especificar casos que propaguen el marcado de datos a través de dichos flujos.

- Al tratarse de análisis de flujo de información mediante lenguajes tipados de seguridad, se obtienen las ventajas de desempeño y completitud en el análisis, pero al mismo tiempo se obtiene como desventaja una baja precisión.

Las ventajas de desempeño obedecen a que el análisis se centra en técnicas de chequeo de tipos(label checking), que al corresponder a etapas de compilación dan como resultado bajo costo en desempeño.

La completitud en el análisis se obtiene haciendo seguimiento al flujo de información de inicio a fin del aplicativo[7], incluyendo tanto flujos implícitos como flujos explícitos, generando así, un menor reporte de falsos negativos.

La baja precisión en el análisis obedece a un enfoque de análisis pesimista, en el que, al incluir todas las posibles ramas de ejecución, se generan más falsos positivos.

- Además de las ventajas y desventajas, el análisis de flujo de información en aplicativos Android por medio del sistema de anotaciones de Jif, comprende varios retos que implican extensiones al sistema de anotaciones de Jif, tanto para soportar clases específicas del framework de Android, como para soportar características del lenguaje Java estándar no soportadas por Jif.

Adicionalmente, para las características del framework Android que definitivamente no se puedan soportar mediante el sistema de anotaciones de Jif, es necesario adoptar mecanismos que permitan analizar el flujo de información en las aplicaciones que las requieran.

- Como conclusión final, es pertinente resaltar que el presente trabajo de investigación explora el análisis de flujo de información de aplicativos Android mediante el sistema de anotaciones de Jif, y que, aunque quedan bastantes retos por superar, el principal aporte es que se posibilita el análisis de flujo de información en aplicativos Android, mediante el sistema de anotaciones de Jif, brindando una herramienta de apoyo al desarrollador para que verifique el cumplimiento de determinadas políticas de seguridad, desde la construcción del aplicativo.

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] A. Documentation. (2015, May) App manifest. [Online]. Available: http://developer.android.com/guide/topics/manifest/manifest-intro.html

[3] ——. (2015, May) Enforcing permissions in androidmanifest.xml. [Online]. Available: http://developer.android.com/guide/topics/security/permissions.html#manifest

[4] (2015, May) App manifest. [Online]. Available: http://developer.android.com/reference/android/content/Intent.html

[5] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *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/

[6] 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.

[7] AndreiSabelfeld and A. wC.Myers, "Language-based information-flow security," *IEEE Jounal*, vol. 21, no. 1, pp. 1–15, January 2003.

[8] Cornell University. (2014, March) Jif reference manual. [Online]. Available: http://www.cs.cornell.edu/jif/doc/jif-3.3.0/language.html#unsupported-java

[9] 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.

[10] C. Fritz, "Flowdroid: A precise and scalable data flow analysis for android," Master's thesis, Technische Universität Darmstadt, July 2013.

[11] 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.

[12] 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.

[13] 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.

[14] B. Yadegari and S. Debray, "Bit-level taint analysis," *2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014.

[15] 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.

[16] 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

[17] J. Graf, M. Hecker, and M. Mohr, "Using joana for information flow control in java programs — a practical guide," *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, pp. 123–138, February 2013.

[18] IBM T.J. Watson Research Center. (2013, July) Walawiki. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page

[19] J. Graf, M. Hecker, and M. Mohr, "Jodroid: Adding android support to a static information flow control tool," *Proceedings of the 8th Working Conference on Programming Languages (ATPS'15)*, February 2015.

[20] Soot. (2012, January) Soot: a java optimization framework. [Online]. Available: http://www.sable.mcgill.ca/soot/

[21] Eric Bodden. (2013, June) Soot: a java optimization framework. [Online]. Available: http://sable.github.io/heros/

[22] Secure Software Engineering Group at EC SPRIDE. (2013, June) Droidbench – benchmarks. [Online]. Available: https://github.com/secure-software-engineering/DroidBench/blob/e64bf483949bf4cb91af642a415fadc9c65e4be5/README.md
[23] Panagiotis Christias. (2015, April) Unix on-line man pages. [Online]. Available: http://dell9.ma.utexas.edu/cgi-bin/man-cgi?00+00
[24] 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.