ANÁLISIS DE FLUJOS DE INFORMACIÓN EN APLICACIONES ANDROID

LINA MARCELA JIMÉNEZ BECERRA

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ 2015

ANÁLISIS DE FLUJOS DE INFORMACIÓN EN APLICACIONES ANDROID

LINA MARCELA JIMÉNEZ BECERRA

Asesores

Martín Ochoa, Ph. D.

Researcher at the software engineering chair of the TU Munich

Sandra Julieta Rueda Rodriguez, Ph. D.

Profesora Asistente, DISC Universidad de los Andes

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍAS

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ 2015

Índice general

Resumen

Breve resumen del trabajo : contexo, problema, solución propuesta, resultados alcanzados.

La presente 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. 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.
Diferentes trabajos de investigación han abordado el problema de pérdida de información en aplicativos Android, sin embargo, la literatura científica existente al respecto, permite inferir que la mayoría de trabajos aplican técnicas para hacer data-flow análisis a partir del bytecode. De modo que, su finalidad es detectar fugas de información y no, verificar que el aplicativo respeta determinadas políticas de seguridad. Así, el desarrollador de la aplicación carece de herramientas de apoyo para verificar si la aplicación que implementa, cumple con determinadas políticas de seguridad.

CAPÍTULO 1

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 están enfocadas a analizar aplicaciones de terceros, asumiendo que el atacante provee bytecode malicioso. Por tanto, aplican data-flow analysis partiendo del bytecode. Estas propuestas no abordan el problema del lado del desarrollador, analizando flujos de información de la aplicación para verificar el cumplimiento de políticas de confidencialidad.
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.

1.1. Técnicas de análisis de código

1.1.1. 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.

Aunque los resultados del análisis estático pueden perder precisión, la ventaja es que son generalizables, porque el modelo construido representa una descripción del comportamiento del programa, independientemente de las entradas y el contexto en que este se ejecute. Ahora, con el análisis dinámico, no es posible generalizar sus resultados para futuras ejecuciones, porque no existen garantías de que las entradas con que fue ejecutado el programa, contengan características para todos los posibles caminos de ejecución.
Además de las ventajas y desventajas de ambas clases de análisis, cada uno implica su propio reto. Mientras en el análisis estático la dificultad está en construir el modelo de abstracción adecuado, en el análisis dinámico, es complejo encontrar un conjunto de casos de prueba representativo.
Por otra parte, dependiendo de la finalidad con que se detecte la fuga de información, un tipo de análisis puede ser más apropiado que otro. Si se busca contener la fuga de información a tiempo de ejecución, análisis dinámico es el camino apropiado. De lo contrario, si se busca garantizar que a tiempo de ejecución la aplicación no incurre en fugas de información, resulta más conveniente aplicar análisis estático, porque cumplir con tales garantías implica definir políticas de confidencialidad y/o integridad desde la implementación de la aplicación. REFERNCIA

Precisamente, el propósito fundamental del presente trabajo es ofrecer al desarrollador de aplicaciones Android una herramienta para aplicar políticas de confidencialidad en la aplicación que implementa, así, la aplicación se ejecutará exitosamente, si y sólo si, cumple con las políticas definidas, de lo contrario, el desarrollador puede revisar y corregir su código.

1.1.2. Detectar o garantizar políticas de seguridad

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. 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).

Acorde a literatura científica en el ámbito de seguridad de aplicativos Android, parte importante de las propuestas para análisis de fuga de información(TaintDroid[4], Flow-Droid[5], DidFail[6], DroidForce[7]), parten del bytecode para realizar data-flow analysis, mediante técnicas de análisis tainting. Las técnicas de análisis tainting, son un tipo especial de análisis de flujo de datos, donde se hace seguimiento al flujo de datos entre un conjunto de fuentes consideradas privadas y/o sensibles; y un conjunto de destinos considerados no confiables, sources y sinks, respectivamente.
Si bien, tales propuestas permiten detectar flujos de datos indebidos en aplicaciones Android, están enfocadas a analizar aplicaciones ya implementadas, y no, en garantizar el cumplimiento de determinadas políticas de seguridad desde su construcción.

Precisamente, mediante análisis basado en control de flujo de información y técnicas Security-Typed, es posible garantizar el cumplimiento de políticas de seguridad en las aplicaciones que se implementa, puesto que, las reglas para evaluar control de flujo de información pueden definirse mediante técnicas Security-Typed, por ejemplo como se definen con Jif 2.1.1, un lenguaje tipado de seguridad para realizar control de flujo de información en aplicativos Java.

1.1.3. Security Typed Languages

En general, 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[8].
Una importante diferencia entre confidencialidad e integridad, es que la integridad de la información de un programa puede ser alterada sin la interacción con agentes externos.
Ambas políticas son fundamentales para garantizar propiedades de seguridad.
Con políticas de confidencialidad, es posible garantizar ausencia de fugas de información. Con políticas de integridad, la finalidad es evitar modificación de la información, de forma no consentida.
Verificar que un programa utilice la información acorde a tales políticas, implica analizar sus flujos de información de inicio a fin. Para tal análisis se deben definir: políticas de flujo de información y controles de flujo de información, es decir, las políticas de seguridad a evaluar y los mecanismos para aplicarlas.

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.

CAPÍTULO 2

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 están accediendo la información, es decir, si información de usuario manipulada por una determinada aplicación, realmente debería ser accedida por otros aplicativos del dispositivo, aún cuando sean considerados no maliciosos; y qué garantías puede ofrecer el desarrollador para que tal acceso, efectivamente sea consentido.

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 tal 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[4], Flow-Droid[5], DidFail[6], DroidForce[7]), indica que la mayoría de propuestas hacen data-flow analysis mediante técnicas de análisis tainting, partiendo del bytecode. Una característica sobresaliente entre estos trabajos es el modelo de ataque, puesto que, se centran en analizar aplicaciones de terceros asumiendo que el atacante provee bytecode malicioso.
Analizar aplicaciones propias para garantizar políticas de confidencialidad e integridad, bajo tales propuestas 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 dinámicamente, la propagación del marcado de datos se centra únicamente en los 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[9][10].
Por el otro, aún cuando se hace análisis tainting estáticamente, 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 la marca para flujos implícitos[11]. Generalmente, las propuestas que basan su análisis data-flow estático, se concentran en propagar el marcado de datos a través de flujos explicitos, y las propuestas mencionadas anteriormente, no son ajenas a tal generalidad(DidFail[6][page 33], FlowDroid[5][page 30]).
Otra razón fundamental para no analizar aplicaciones propias con tales propuestas es que están diseñadas para detectar flujos indebidos de datos, en aplicaciones ya construidas, y no, para garantizar el cumplimiento de políticas de seguridad en una aplicación desde su construcción.
Los riesgos de seguridad tras el under-tainting de datos, y la ausencia de garantías en el cumplimiento de determinadas políticas de seguridad, pueden superarse mediante control de flujo de información, Information Flow Control(IFC), puesto que, con esta técnica se analiza estáticamente la aplicación para identificar todos los posibles caminos que podrían tomar sus flujos de información, garantizando que a tiempo de ejecución, la aplicación respeta políticas de seguridad.

Finalmente, partiendo del contexto que se plantea, dónde se cuenta con el código fuente Android, porque es el propio desarrollador quien requiere evaluar políticas de confidencialidad en su aplicación, para garantizarle al usuario que la aplicación las cumple. Resulta apropiado proveer una herramienta de apoyo al desarrollador, mediante la cual analice el flujo de información de la aplicación próxima a liberar, y verifique el cumplimiento de políticas de seguridad.

2.1. Trabajos Relacionados

2.1.1. JIF

JIF(Java Information Flow), es un lenguaje tipado de seguridad que permite extender el lenguaje de programación Java, con control de flujo de información y control de acceso, usando anotaciones de seguridad. El compilador usa estas anotaciones durante el chequeo de tipos, verificando el cumplimiento de la propiedad de seguridad non-interference[12].

Usar JIF para el análisis estático de flujo de información de un programa, requiere implementar la versión del mismo, especificando mediante el conjunto de labels de JIF, las políticas de seguridad a verificar. La implementación de programas JIF está basada en el modelo de etiquetas DLM(Decentralized Label Model), donde un principal es una entidad con autoridad para observar y cambiar aspectos del sistema, así, un principal puede definir y hacer cumplir los requerimientos de seguridad del dueño de la información. Para expresar una relación de confianza entre principals, se define la relación acts-for, a partir de la cual, se derivan dos tipos de principals: top principal y botton principal, un top principal puede actuar para todos los principals, mientras que, un botton principal permite que todos los principals actúen para el. Las políticas de seguridad se condensan en Políticas de Confidencialidad y Políticas de Integridad, con ellas se determina el conjunto de principals readers y writes, y el comportamiento que deberían tener. El compilador de JIF aplica chequeo de labels para verificar el cumplimiento de las políticas de seguridad definidas en el programa, cuando determina que efectivamente las cumple, da paso al compilador de Java para generar su versión ejecutable.

Además del modelo de labels en que se centra, JIF incluye mecanismos que aportan características adicionales en la implementación de programas para seguimiento de Flujo de información. La opción de flexibilizar las políticas de seguridad de la información, hace parte de estas características adicionales, y se logra aplicando el mecanismo Downgrading. Dependiendo del tipo política al que se realiza downgrading, políticas de confidencialidad o políticas de integridad, el proceso se conoce como Declasificación o Endorsement, respectivamente.

2.1.2. JOANA

JOANA (Java Object-sensitive ANAlysis)- Information Flow Control Framework for Java[13]. 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[14], 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.

2.1.3. JoDroid

JoDroid[15] 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.

2.1.4. 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[2.1.7], un clasificador automático de sources y sinks para la Api de Android.
FlowDroid provee un alto recall y precisión[5] 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[16], un framework que genera código intermedio para código Java y código ejecutable Android(dex). Adicionalmente, con el framework Heros[17], 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[18], 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[5, 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.

2.1.5. 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 que permiten salir 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 mar-

cas 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 no hacer seguimiento a flujos de control, generando no detección de flujos implícitos[4, 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.

2.1.6. Comparación de técnicas

Las técnicas utilizadas para análisis de seguridad en aplicaciones, pueden aplicarse estática o dinámicamente, dependiendo de las propiedades del programa en que se centre el análisis.
La ejecución dinámica o estática del análisis, trae sus propias ventajas y desventajas. En el caso de análisis estático, completitud en el análisis es una de sus principales ventajas. Esto debido a qué, el análisis contempla todas los caminos de ejecución en que podría incurrir el programa. Evitando que se pierdan casos a analizar. Por otra parte, al carecer de información que sólo se puede obtener a tiempo de ejecución, por ejemplo, las entradas que el programa recibe, el análisis estático suele generar falsos positivos.
En el análisis dinámico, una de las principales ventajas es la baja generación de falsos positivos, puesto que, el análisis no se centra en los posibles casos de ejecución, sino que verifica el caso de ejecución que efectivamente está ocurriendo. No obstante, el análisis dinámico podría incurrir en incompletitud, porque sólo verifica los casos de ejecución que se presenten, es decir, el aplicativo podría presentar fugas de información no reportadas por el análisis, como consecuencia de la no ejecución de los casos que permiten identificarlos.
Así, el análisis dinámico genera menor cantidad de falsos positivos que el análisis estático, sin embargo, el análisis estático ofrece mayor completitud en el análisis.
Adicional a la forma en que son aplicadas, estática o dinámicamente, las técnicas de análisis pueden enfocarse en **analizar el flujo de datos**, o en **analizar flujo de información**.
Las técnicas basadas en tanting análisis, permiten hacer análisis de flujo de datos, marcando los datos de interés y verificando su flujo entre sources(fuentes del programa consideradas sensibles y/o confidenciales) y sinks(destinos considerados no confiables). Entre las desventajas de está técnica, esta el under-tainting, es decir, la posibilidad de fugas a través de datos no marcados para el análisis.

Las **técnicas para hacer análisis de flujo de información**, se enmarcan en técnicas de seguridad de tipado(Security-Typed Analyses), o en técnicas de grafos de dependencia: System Dependence Graphs(SDG).
Las técnicas de security Type, verifican el cumplimiento de políticas de seguridad mediante etapas de compilación(label checking), estás técnicas suelen ser más rápidas pero menos precisas porque generalmente no ofrecen característica de flow-sensitive, context-sensivite y object-sensitive, dando lugar a la generación de falsos positivos. Al mismo tiempo, estás técnicas tienen la ventaja de detectar mayor cantidad de fugas de información(presentan menos falsos negativos).[19]
Las técnicas basadas en grafos de dependencia son flow-sensitive, context-sensivite y object-sensitive, dando mejor precisión en el análisis. Sin embargo, su ejecución es costosa, ya que genera una complejidad de orden polinomial, O(N)3[20, page 3].

Si bien, ambas técnicas utilizan anotaciones en el código del aplicativo de análisis, la forma de anotación es diferente. Por un lado, con lenguajes tipados de seguridad se provee todo un sistema de anotaciones para implementar aplicativos con determinadas políticas de seguridad. Por el otro, en grafos de dependencia sólo se anota el nivel de seguridad de la información a evaluar en aplicativos ya implementados. Con lenguajes tipados de seguridad el desarrollador tiene mayor control sobre la anotación de sus aplicativos.

Las motivaciones para guiar el análisis bajo una u otra técnica, implica poner a consideración tanto el nivel de precisión requerido por las propiedades de seguridad a evaluar, como el costo de implementación y de ejecución del análisis.

2.1.7. Clasificación de Sources y Sinks

En el ámbito de análisis de flujo de información de aplicaciones, independientemente del tipo de análisis, estático o dinámico, el punto de partida es la definición de políticas de privacidad, los pasos sucesivos para detectar la perdida de información giran en torno a las políticas de privacidad definidas. 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. 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 recibe como entrada métodos de Android y devuelve una lista con la respectiva categorización de sources y sinks.
La construcción del modelo de análisis propuesto, parte definiendo los elementos necesarios para el reconocimiento de sources y sinks; inicialmente define: Sources y sinks, respectivamente, como las entradas y salidas de flujo de datos del programa; un dato como un valor o una referencia a un valor; un Resource Method como un método que lee o escribe datos en un recurso compartido. Seguidamente, define el concepto de sources y sinks, considerando el contexto de Android: Android Sources

como llamadas a métodos tipo resources(Resources method) que retornan valores no constantes al código de la aplicación. Android Sinks como llamadas a methods resource, aceptando como argumento al menos un valor no constante desde el código de la aplicación, y qué además adicionen o modifiquen valores del recurso invocado.
El modelo de entrenamiento de SuSi usa el clasificador SMO, una implementación del clasificador SVM(Support Vector Machines) para Weka, al que inicialmente enseña a clasificar partiendo de ejemplos entrenados manualmente. Adicionalmente, lo adapta utilizando la técnica de clasificación one-againts-all, de modo que pueda representar, tanto los ejemplos de entrenamiento, en tres clases: sources, sinks, o ninguno; como las categorías de los sources y sinks identificados.
Los criterios de clasificación están basados en un conjunto de características, es decir, funciones que asocian ejemplos de entrenamiento o ejemplos de prueba, con un determinado valor.
El proceso de análisis se compone de dos rondas secuenciales: clasificación y categorización. Cada una se compone de las fases input, preparation, classification y output. Así, la salida de la primera ronda: 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.

2.2. Background

2.2.1. Aplicaciones Android

Una aplicación Android puede estar compuesta 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 Receives reciben mensajes envíados por el sistema o por otras aplicaciones.

2.2.2. Sistema de anotaciones en Jif

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)[21], verificando que los flujos de información generados cumplen con las restricciones establecidas.

2.2.3. DML(Decentralized Label Model)

Jif basa su sistema de anotaciones en el modelo de etiquetas DLM(Decentralized Label Model), donde se manejan tres elementos fundamentales: Principals, Políticas y Labels.
Principals: 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}.
Labels: un label consiste en un conjunto de políticas de confidencialidad e integridad. Los labels se escriben en las expresiones del programa que se anota(labels de seguridad), esto es métodos, variables, arrays, etc..
En síntesis, las políticas de seguridad definen que principals pueden leer o modificar la información, y esas políticas se expresan mediante labels.

2.2.4. Label Checking

Para hacer seguimiento al flujo de información de un programa, el compilador de Jif asocia un label al program counter de cada punto del programa, progam-counter label($\underline{pc}$). En cada punto del programa, el ($\underline{pc}$) representa la información que podría conocerse tras la ejecución de ese punto del programa. El ($\underline{pc}$) es afectado por los labels 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 los labels que han afectado el flujo de información para llegar a un determinado punto de ejecución.
Adicionalmente, jif define labels que representan la información que podría conocerse tras la terminación normal, o terminación por excepción de las sentencias del programa. Y labels enviroments, que para cada punto del programa determinan la forma en que se relacionan labels y principals.
El valor de dichos labels es verificado durante la compilación del programa, si se detecta que no cumplen con las restricciones establecidas en la anotación del mismo, el compilador genera error, indicando los puntos del programa que las incumplen.

2.2.5. Sintaxis de Anotación en 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} el label de seguridad para especificar quien es el dueño de la variable, y varName, el respectivo nombre de la variable.

Definición de arrays

En Java un array se define de la forma:

```
modifier java-type [ ] nameArray
```

En Jif, además del tipo de dato Java(java-type) de los elementos almacenados en el array, se deben especificar dos labels de seguridad: Base Label(BL) y Size Label(SL). BL indica el nivel de seguridad de los elementos que almacena el array, controlando quien puede conocer la información del mismo. SL especifica quienes pueden conocer la número de elementos almacenados. Así, la sintaxis para anotar el array es:

```
java-type {BL} [ ]{SL} nameArray
```

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 se debe asociar un label 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). 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 el label de seguridad para el valor devuelto por el método.
BL, Begin Label, representa el máximo nivel se seguridad del $\underline{pc}$ desde donde se invoca el método, de este modo, el program counter label 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, los labels 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 $\underline{pc}$ en el punto de terminación del método, y representa el máximo nivel de información que puede conocerse tras la finalización del método.

2.2.6. 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.

2.2.7. 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:})

CAPÍTULO 3

Diseño y Especificaciones

3.1. Propuesta de solución

La propuesta para detectar fuga de información en aplicaciones Android, antes de su publicación, consiste en proveer al desarrollador una herramienta para análisis estático de flujos de información de la aplicación. Así, partiendo de las anotaciones de seguridad que el desarrollador defina en el código fuente, se verifica si la aplicación cumple con políticas de seguridad.
Los requerimientos iniciales para construir tal herramienta son: un lenguaje tipado de seguridad que permita anotar código fuente Android, y el conjunto de reglas que evaluarán las políticas de seguridad.
Al consultar literatura científica al respecto, se encuentran herramientas como Jif 2.1.1 y Joana 2.1.2, especializadas en anotar código Java, pero no código Android. Es decir, las anotaciones son válidas para clases del lenguaje java estándar, pero no para clases específicas de la API de Android.

Si bien, ambas herramientas analizan flujos de información en aplicaciones Java, y podrían ser extendidas para anotar código Android, sus técnicas de análisis y forma de anotación son diferentes. Por un lado, Jif es un lenguaje tipado de seguridad que basa su análisis en el chequeo de tipos. Por el otro, Joana es un framework basado en análisis de grafos de dependencia, enfocado a precisión.
Mientras Jif se basa un un modelo de anotaciones(DML), permitiendo la implementación de aplicativos con políticas de seguridad; Joana sólo requiere anotaciones para el nivel de seguridad de la información a analizar, en aplicativos ya implementados.
Adicionalmente, el modelo de anotaciones (DLM) de Jif, define la lattice de seguridad adecuada para las anotaciones en el código fuente, ofreciendo un maduro sistema que además de evaluar políticas de confidencialidad, e integridad, permite definir características de seguridad adicionales como declasificación y endorsement.
Acorde a los propósitos del presente trabajo, Jif ofrece los beneficios de un lenguaje tipado de seguridad y un sistema sólido de anotaciones, facilitando la definición de las propiedades de seguridad a verificar.

Partiendo de Jif como el lenguaje tipado de seguridad, los retos subsiguientes son: implementar el setup de Jif para Android e integrar un clasificador para sources y sinks de Android.

El setup de Jif para Android consiste en implementar adaptaciones necesarias a las clases de la API Android, de modo que, el compilador de Jif reconozca anotaciones Jif en aplicaciones Android. Estas adaptaciones son necesarias porque, aunque Jif permite extender al lenguaje Java, y en el fondo las clases de la API Android están implementadas en Java, si no se cuenta con una versión Jif de tales clases, el compilador de Jif no tiene como reconocerlas.
Con la integración de un clasificador de sources y sinks para Android al sistema de anotaciones de JIF, se provee información necesaria para construir las políticas de seguridad. Porque, al conocer qué código de la API de Android, es considerado como source o como sink, se tiene el criterio para decidir su anotación.

Básicamente, se requiere un módulo que extienda las clases en JIF para que el lenguaje reconozca código de la API Android(*Android Jif Setup*), más un módulo que integre el clasificador de sources y sinks para Android(*Sources y Sinks*). Ambos módulos deben tener comunicación con el módulo que evalúa las políticas de seguridad *verificador de políticas*, que en esencia es el compilador de Jif.
Adicionalmente, la herramienta debe recibir como entrada el código fuente de la aplicación, debidamente anotado por el desarrollador. De modo que el desarrollador defina las políticas de seguridad a evaluar. A partir de tales anotaciones, la herramienta de análisis verifica si los flujos de información del aplicativo, cumplen con la política de seguridad expresada a través sus anotaciones, y retorna los resultados del análisis.

Habiendo realizado las extensiones necesarias, se espera contar con una herramienta de análisis de flujo de información, para aplicativos Android, mediante el sistema de anotaciones de Jif.

3.2. Diseño de la solución

Como se describe en la propuesta de solución 3.1, 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 manera que, la herramienta evalúe flujos de información en aplicativos Android, para verificar el cumplimiento de las políticas de seguridad que el desarrollador define en el código, mediante el sistema de anotaciones de Jif.
Sin embargo, para efectos de esta tesis se limita el Setup de Jif, partiendo de una política de seguridad específica. Así, el diseño se centra en soportar un conjunto reducido de clases de la API de Android(Anotaciones a la API), 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 2.1.7.
Adicionalmente, para aspectos de evaluación se incluye el diseño de un anotador. De modo que, acorde a la política de seguridad establecida, se genere la versión anotada del aplicativo a analizar(la cual debería ser proveída por el desarrollador).
La figura 3.1 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,

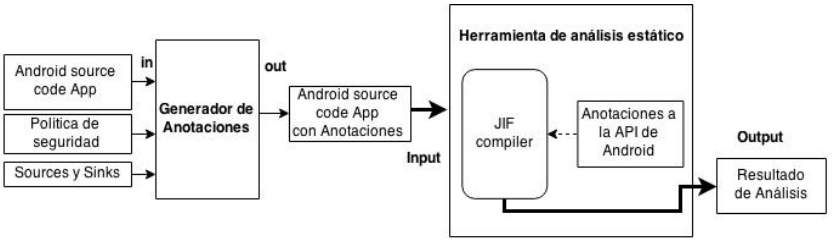

Figura 3.1: Diseño herramienta de análisis estático. El generador de anotaciones retorna la versión anotada del aplicativo a analizar, partiendo del código fuente del aplicativo Android, la política de seguridad a evaluar, más los sources y sinks requeridos para verificar la política. La herramienta de análisis estático está integrada por el compilador de Jif, más anotaciones a la API de Android. Esta recibe el aplicativo Android debidamente anotado y retorna el análisis de flujo de información.

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(verificador de políticas) y las anotaciones a la API de Android. Con anotaciones a la API se hace referencia a clases de la API android que deben ser reconocidas por el compilador de Jif, para verificar el cumplimiento de la política de seguridad establecida. Cabe anotar que, tal reconocimiento no implica modificaciones al compilador de Jif.

Siguiendo el esquema de diseño anteriormente descrito: primero, se define la política de seguridad a evaluar 3.3; segundo, se toman a consideración elementos influyentes para verificar el cumplimiento de la política mediante Jif 3.4; y tercero, teniendo en cuenta 3.3 y 3.4, 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).

3.3. Definición de la política de seguridad

Detectar si una aplicación Android(perteneciente al conjunto evaluable) presenta flujos de información entre, información con nivel de seguridad alto e información con nivel de seguridad bajo.
Detectando fugas de información catalogada con nivel de seguridad alto, vía: canales creados durante el control de flujo del programa(flujos implícitos), mensajes de texto y mensajes de Log.

3.4. Consideraciones para verificar el cumplimiento de la política mediante Jif

Teniendo definida la política de seguridad a verificar se describe: cuál es la información considerada con nivel de seguridad alto; cuáles son los canales que muestran

información con nivel de seguridad bajo; cuál es el punto de entrada para el análisis de la aplicación; qué se espera del desarrollador si la aplicación a analizar requiere excepciones tipo Runtime; qué se asume para evaluar el flujo de información; y cómo es el acceso a métodos que deben ser sobreescritos.
Todos, elementos relevantes para proponer un esquema de anotación.

Información considerada con nivel de seguridad alto
Para verificar el cumplimiento de la política de seguridad a evaluar se parte de un conjunto de sources, caracterizados por dar a conocer información del usuario, considerada como privada o sensible. Los métodos que integran el conjunto de sources son: 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 referencia corresponde a un campo tipo textPassword, campos destinados a almacenar contraseñas.
Este conjunto de sources es tomado del listado proveído por el clasificador SuSi 2.1.7.

Canales que muestran información con nivel de seguridad bajo
La información enviada a través de mensajes de texto y la información conocida tanto a través de mensajes de Log, como a través de canales generados por el control de flujo del programa, tiene en común que debe poderse conocer por terceros. En consecuencia, se considera que estos canales deben dar a conocer información con nivel de seguridad bajo.
En el caso de mensajes de texto y mensajes de Log, se hace referencia específicamente a las clases Log y SmsManager de la API de Android.

Diferencia entre una aplicación Android y una aplicación Java convencional
En esencia, una aplicación Android es una aplicación Java con interfaces descritas en XML, que para ser ejecutada necesita del framework de Android, porque este le provee acceso al hardware del dispositivo y funcionalidades del sistema.
Por otro lado, Jif permite hacer seguimiento al flujo de información de una aplicación Java, extendiendo el lenguaje mediante labels de seguridad.
Para analizar flujo de información de una aplicación Android mediante Jif, es importante mencionar que mientras una aplicación Java convencional cuenta con un único punto de entrada para iniciar su ejecución, esto es, la clase principal donde se implementa el método main; una aplicación Android puede tener más de un punto de entrada, generados a partir de los diferentes tipos de componentes que le integren(Activity, Service, Content Provider y Broadcast Receiver). La necesidad de interacción del usuario para activar tales puntos de entrada varía acorde al tipo de componente, así, en el caso de componentes tipo Activity su ejecución sólo inicia hasta que el usuario interactúe con la actividad, y para manejar dicha interacción, la API Android provee el método OnCreate. De otro modo, componentes tipo Service y Broadcast Receiver, inician su ejecución a través de los métodos OnStartCommand y OnReceive, respectivamente, sin necesidad de interacción del usuario.
Teniendo en cuenta lo anterior, se asume que la aplicación a evaluar tiene un único punto de entrada, que depende del tipo de componente que implemente.

Chequeo de excepciones tipo Runtime
En lenguaje Java las excepciones tipo Runtime tales como NullPointerException, no son verificadas a tiempo de compilación, sin embargo, buscando evitar la generación de canales encubiertos mediante las mismas, Jif si las verifica. En consecuencia, si un

programa requiere excepciones NullPointerException, ClassCastException y/o ArrayIndexOutOfBoundsException, el programador debe declararlas en el programa, de lo contrario, el compilador de Jif genera error. Para las aplicaciones a analizar, se espera que el desarrollador haya especificado las excepciones necesarias.

Evaluación del flujo de información
Para evaluar el flujo de información, se asume que todos los métodos definidos en la clase serán invocados, y por tanto, todos son incluidos en el análisis.
Esta decisión de análisis busca evitar el paso inadvertido de flujos de información, generados por omisión.

Acceso a métodos de sobreescritura.
Los métodos de las clases Activity, Service y BroadcastReceiver, son métodos que se pueden sobreescribir, todo programa Android que extienda de tales clases debe poder utilizarlos.

3.5. Lineamientos de anotación

Los lineamientos de anotación definen los elementos básicos de anotación 3.5.1, anotaciones necesarias para la API de Android 3.5.2 y anotaciones en los aplicativos a analizar 4.2.

3.5.1. Elementos básicos de anotación

En 3.4 se definió qué *Canales muestran información con nivel de seguridad bajo* y cuál es la *Información considerada con nivel de seguridad alto*. Ahora, para anotar la información catalogada con uno u otro nivel de seguridad, 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, lo primero que se debe definir es quién es la autoridad de los programas y cuáles son los labels de seguridad.
Conceptos y términos implicados en la sintaxis de anotación Jif, se encuentran en la sección de background 2.2.

Principal *Alice*
haciendo uso de los principals 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.

Label de seguridad {*Alice:*}
este label 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.

Label de seguridad {}
este label indica que la información tiene nivel de seguridad bajo, es decir, información de conocimiento público.

3.5.2. Anotaciones a la API de Android

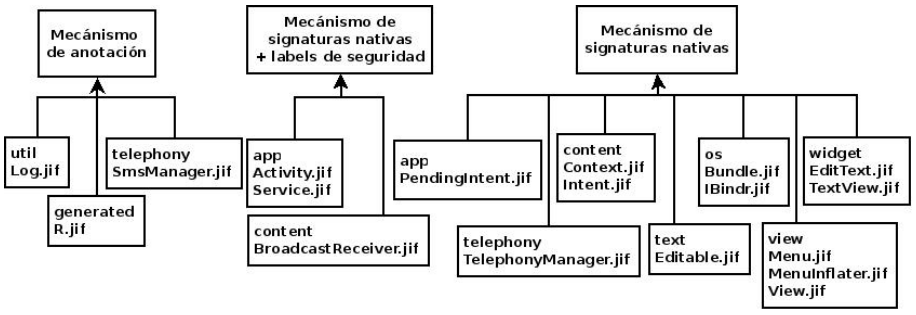

Figura 3.2: Mecanismos de anotación para clases de la API.

Anotaciones para Canales que muestran información con nivel de seguridad bajo

Para controlar el flujo de información que se envía hacia *Canales que muestran información con nivel de seguridad bajo*, 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.
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 mostrar, y por tanto esa información debe ser pública.
Por consiguiente, en la definición del método el label de seguridad del argumento(AL) correspondiente a la información a mostrar(logs) o la información a enviar(sms), se anota con el label de seguridad bajo(label público{}). Con esto se garantiza que la información se muestra o envía, si y sólo si, el nivel de seguridad del argumento con que se invoca el método es público. Por ejemplo, si el método se invoca con información anotada con label de seguridad alto, se genera error en la compilación del programa que le llama.
El resto de argumentos del método se anotan con label de seguridad {Alice:}, puesto que esa información puede ser pública o privada.
En el caso del BL al anotarlo con el label {Alice:}, se permite que el método sea invocado desde cualquier punto de un programa. Para el resto de labels se dejan los que Jif genera por defecto.
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) { }
```

Anotaciones para métodos de sobreescritura

En 3.4, también se definieron las clases de la API para las que se debe soportar el *Acceso a métodos de sobreescritura*(Activity, Service y BroadcastReceiver). La anotación para la definición de tales métodos se basa en: (1)reglas de Jif para la sobreescritura de métodos y (2)desde dónde pueden ser invocados. (1)Jif exige que el nivel de seguridad del BL del método desde donde se invoca el método a sobreescribir, no debe ser menos restrictivo que el BL de la definición de tal método(privado es más restrictivo que público). (2)los métodos a sobreescribir deben poder invocarse desde aplicativos que extiendan de las clases Activity, Service y BroadcastReceiver. Para cumplir con (1) y (2), los métodos que requieren ser sobreescritos se definen con BL público({}). De este modo los aplicativos desde dónde se invocan los métodos a sobreescribir deben tener igual BL.
Por ejemplo, el método OnCreate de la clase Activity, se define de la forma:

```
protected native void onCreate{}(Bundle savedInstanceState);
```

Anotaciones a librerías de la API

adicional a las clases Log, SmsManager, Activity, Service y BroadcastReceiver, es necesario soportar las librerías requeridas por estas, para lo cual se utilizan signaturas nativas de anotación. Tales librerías son:
android.app.PendingIntent
android.telephony.TelephonyManager
android.content.Context
android.content.Intent
android.text.Editable
android.os.Bundle
android.os.IBinder
android.widget.EditText
android.widget.TextView
android.view.Menu
android.view.MenuInflater
android.view.View

Integración de clases de la API de Android a las clases reconocidas por el compilador de Jif

Definidos los criterios de anotación para las clases de la API, a las cuales se debe implementar su respectiva versión Jif, se definen los mecanismos a utilizar para implementar tal versión. Además del mecanismo de anotación completa en que se basa la implementación de aplicativos en Jif(mecanismo de anotación[1]), el compilador provee un mecanismo que permite interactuar con código de clases Java ya existentes[22], 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

[1]Para hacer referencia a los mecanismos con que se da soporte a las diferentes clases de la API, se adoptan los nombres: mecanismo de anotación, mecanismo de signaturas nativas y mecanismo de signaturas nativas más labels de seguridad.

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).
Para el presente caso y de acuerdo a los criterios de anotación previamente establecidos las clases a implementar mediante uno u otro mecanismo, son ilustrados en la figura 3.2.

3.5.3. Anotaciones en los aplicativos a analizar

Los elementos en que se fundamenta el esquema de anotación para el aplicativo a analizar, son los siguientes:
Primero, como los *Canales que muestran información con nivel de seguridad bajo* están anotados desde la definición en sus respectivas clases de la API, de modo que la información a mostrar(logs) o enviar(msn) debe tener nivel de seguridad bajo(debe ser pública), el esquema de anotación del aplicactivo debe proveer los labels de seguridad adecuados a la información con que se invocan tales canales.

Segundo, la anotación para la sobreescritura de métodos de la API, está definida en las respectivas clases de la API, el esquema de anotación para la invocación de los métodos a sobreescribir debe ser consecuentes con tales definiciones de anotación.

Tercero, el esquema de anotación para el aplicativo a analizar parte de los sources que se identifiquen en el mismo. Tales sources pertenecen al conjunto definido en (*Información considerada con nivel de seguridad alto*), conjunto que está integrado por el tipo de dato EditText[2] y los métodos: getDeviceId, getSimSerialNumber, findViewById, getLatitude, getLongitude y getSubscriberId.

Cuarto, se parte de los *Elementos básicos de anotación* previamente definidos. Así, una clase Android tendrá como principal(autoridad máxima) al principal *Alice*, y los labels de seguridad para anotar información con nivel de seguridad alto y nivel de seguridad bajo son: {*Alice:*} y {}, respectivamente.

Estructuración del esquema de anotación

Para concretar el esquema con que se anotan los aplicativos a analizar, se establece la finalidad(Objetivo de la anotación), qué se va a anotar(Elementos del esquema), cómo se van a anotar tales elementos(Anotación de los elementos) y finalmente se indica cómo aplicar el esquema de anotación(Pasos para aplicar el esquema de anotación). Así:

Objetivo de la anotación
El esquema de anotación se centra en identificar si una clase contiene sources, y verificar si los métodos de la clase influencian esa información, de modo que, cuando la información sea enviada por canales con nivel de seguridad bajo, tenga el nivel de seguridad adecuado.

[2]Este 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.

Elementos del esquema
Para referenciar los elementos que se anotan mediante este esquema, se establecen una serie de términos, así:
- Variable source: una variable source es una variable que almacena información proveída por algún elemento del conjunto de sources.
- Array source: un array source es un array que almacena información de variables source.
- Método que se sobreescribe: hace referencia a métodos de la API android que deben ser sobreescritos para la implementación del aplicativo, por ejemplo, el método Oncreate que se sobreescribe al implementar componentes tipo Activity.
- Método source: un método source es un método(definido dentro de la clase) que cuando es invocado, contiene dentro de los parámetros de invocación, al menos una variable source.
- Método no source: un método no source es un método(definido dentro de la clase) que cuando es invocado, no contiene dentro de los parámetros de invocación, variables source.

Anotación de los elementos
Fijados los elementos que se anotan, se define su respectiva forma de anotación. Conceptos y términos implicados en la sintaxis de anotación Jif, se encuentran en la sección de background 2.2.5. Los labels de seguridad que no son anotadas mediante este esquema, son generados automáticamente por Jif, acorde a los labels que estable por defecto para la definición de variables, métodos y arrays, etc.

- Definición A: anotación de variable source.
java-type {*Alice:*} *nameVar*
como la información almacenada en una variable source es información con nivel de seguridad alto(conjunto de sources), la variable debe tener un label de seguridad que indique que su información es de alta confidencialidad. Al anotarla con el label {*Alice:*}, se está indicando que esa información pertenece al principal *Alice*, así, cuando se envíe hacia un canal de seguridad bajo, da lugar a un flujo de información indebido(de nivel alto a nivel bajo).

- Definición B: anotación de array source.
java-type{*Alice:*}[ ]{*Alice:*} *arrayName*
debido a que un array source almacena información con nivel de seguridad alto, se debe garantizar que sólo el principal dueño de la información pueda conocer tanto items como tamaño del array. Esto se logra anotando con label de seguridad {*Alice:*} el *BL* y *SL*, labels de seguridad para los elementos y tamaño del array, respectivamente.

- Definición C: anotación de método que se sobreescribe.
El BL de un método a sobreescribir debe ser anotado con label de seguridad bajo {}, puesto que en su respectiva definición en las clases de la API, han sido definidas con ese BL.

- Definición D: anotación de método source.
java-type{*Alice:*} *methodName* {*Alice:*}*(java-type arg1*{*Alice:*} *,,, java-type argn*{*Alice:*}*)*
Como un método source influencia información con nivel de seguridad alto(variable source), se debe garantizar que sólo sentencias del programa que tengan nivel de seguridad alto, puedan ser actualizadas con información proporcionada por el método.

Buscando cumplir con lo anterior, se anota con label de seguridad {Alice:} el RTL y el BL del método. Retomando la sintaxis jif para anotación de métodos:
java-type {*RTL*} *methodName* {*BL*} *(java-type arg1*{*AL*}*,,, java-type argn*{*AL*}*):*{*EL*}
Al anotar {RTL} con {Alice:} se asegura que, si el método retorna un valor, tendrá nivel de seguridad alto.
Al anotar {BL} con {Alice:} se asegura que, los puntos del programa que traten de ser actualizados tras el llamado del método(por ejemplo variables, o resultados de condicionales) deben tener un pc con nivel de seguridad alto.
Para el caso de los AL, labels de seguridad de los argumentos, el sistema de anotaciones de Jif exige que los AL con que se invoca el método, deben ser igual o menor de restrictivos que los AL con que se define el método. Por otro lado, se tiene certeza de que uno de los argumentos con que se llama el método tiene AL con nivel de seguridad alto(pues ese fue el criterio con que se clasificó al método como source), pero el resto de parámetros puede tener AL alto o bajo, entonces para garantizar que el método pueda ser invocado bajo tales condiciones, los AL para los argumentos del método son anotados con label {Alice:}.

- Definición E: anotación de método no source.
methodName {} *(java-type arg1*{*Alice:*}*,,, java-type argn*{*Alice:*}*)*
Como el método no recibe la variable source, el nivel de seguridad de las sentencias del programa que se actualicen con la información del método puede ser bajo. Ahora, si en el cuerpo del método se define o actualiza información con nivel de seguridad alto(información global), tales sentencias deben tener nivel de seguridad alto. Anotando el BL con label {}, se respeta tal condición puesto que, el BL se ve afectado cuando el cuerpo del método tiene información con mayor nivel de seguridad, obligando a que las sentencias a actualizarse con la información del método tengan nivel de seguridad alto.
Para el caso de los AL, labels de seguridad de los argumentos, el sistema de anotaciones de Jif exige que los AL con que se invoca el método, deben ser igual o menor de restrictivos que los AL con que se define el método. Anotando los AL de la definición del método con label {Alice:}, se cumple tal restricción ya que el método es invocado con parámetros cuyo nivel de seguridad es bajo.

Pasos para aplicar el esquema de anotación
Partiendo de las anteriores definiciones, los pasos para la anotación son los siguientes:
(1) Identificar variables source de la clase. Si se encuentran variables sources continuar con los pasos (2) a (11), sino, continuar con pasos: (3), (6), (9) y Aplicar Definición E a los métodos que no se sobreescriben[3]. (2) Identificar arrays sources de la clase. (3) Identificar el total de métodos de la clase.
(4) Del total de métodos listar los métodos source.
(5) Del total de métodos listar los métodos no source.
(6) Del total de métodos listar los métodos a sobreescribir
(7) Aplicar Definición D a listado del paso(4).
(8) Aplicar Definición E a listado del paso (5).
(9) Aplicar Definición C a listado del paso (6).
(10) Aplicar Definición B a listado del paso (2).
(11) Aplicar Definición A a listado del paso (1).

[3]Como no se identifican sources, la anotación de método no source(Definición E) es aplicable para los métodos que no se sobreescriben.

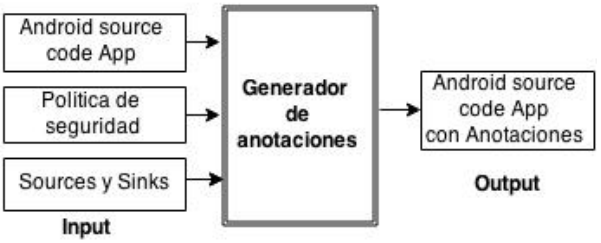

Figura 3.3: Entradas y salidas para el generador de anotaciones.
Para generar la versión anotada del aplicativo a analizar, el anotador parte del código fuente del aplicativo, la política de seguridad definida en 3.3 y el conjunto de sources y sinks implicados en la misma.

Para automatizar estos pasos, se debe implementar un generador de anotaciones(prototipo de anotación). Que partiendo del código fuente de la aplicación Android(perteneciente al conjunto evaluable), la política de seguridad previamente definida y el conjunto de sources y sinks; retorne la implementación en Jif del aplicativo a analizar, versión que contiene las anotaciones para evaluar la política de seguridad definida.
La figura 3.3, ilustra las entradas y salidas esperadas.
Luego la versión obtenida se evalúa con el compilador de Jif.

CAPÍTULO 4

Descripción implementación

4.1. Limitaciones técnicas

Antes de iniciar con la implementación del esquema de anotación propuesto, tanto para las clases de la API 3.5.2, como para los aplicativos a analizar 3.5.3, se identifican una serie de limitaciones del lenguaje Jif para anotar código de la API de Android. Tales limitaciones son adicionales a las características del lenguaje Java no soportadas por Jif, a continuación se describen tanto las encontradas, como las especificadas en el manual de referencia de Jif.

4.1.1. Características del lenguaje Java no soportadas por jif

Si bien, el sistema de anotaciones de Jif hace extensiones al lenguaje java, permitiendo la evaluación de políticas de confidencialidad e integridad para aplicativos implementados en dicho lenguaje, el manual de referencia de Jif precisa las características del lenguaje Java no soportadas[21]. Estas son:

- nested classes: clases que son definidas dentro de otras clases.
- initializer blocks: bloques de código declarados dentro de la clase pero sin pertenecer a ningún método, dependiendo de si se trata de static initialization blocks, su código es el primero en ejecutarse, una vez se carga la clase; o si se trata de instance initialization blocks, su código se ejecutan cada vez que se crea una instancia de la clase.
- threads.

Partiendo de estas precisiones, aplicaciones Android que presenten tales características son excluidas del grupo de aplicaciones a analizar(conjunto de aplicaciones evaluables) mediante la herramienta propuesta.

4.1.2. Soporte para la clase java.lang.Override

En la construcción de aplicaciones Android, según el componente que se esté implementando(activities, content providers, receivers, services), se requiere sobreescribir métodos de la clase que extienda el componente. Así, cuando se define un componente tipo Activity, que debe extender de la clase Activiy.java, se sobreescriben métodos como Oncreate. Cada que se sobreescribe un método se utiliza el statement @Override, con el cual se informa al compilador de Java que el método es sobreescrito. No obstante, al implementar la versión Jif de aplicaciones Android con dicho statement, el compilador de Jif no lo reconoce. La dificultad que se presenta está en el reconocimiento del statement(carácter @ y clase Override), y no en la sobreescritura de métodos, puesto que Jif soporta tal característica.
El soporte para la sobreescritura de métodos es confirmado con una sencilla prueba, anotando la clase Activity.java del framework Android (con un único método, el método Ocreate), e implementando la versión Jif de una aplicación Android que extiende de tal clase, en la cual se define una actividad y sobreescribe el método Oncreate. Cuando se comenta la sentencia @Override, el compilador de Jif identifica la sobreescritura del método y reporta comentarios para el flujo de información.

Al investigar el por qué Jif no reconoce tal sentencia, se encuentra que dentro de las clases Java estándar soportadas por el compilador de Jif, no se incluye la clase java.lang.Override.
Las clases Java estándar pertenecientes a los paquetes io, lang, math, net y sql; soportadas po el compilador Jif, vienen implementadas con anotaciones en el directorio sig-src, directorio que forma parte de la distribución del compilador de Jif con que se esté trabajando.

Un mecanismo para permitir el análisis de flujo de información entre métodos que se sobreescriben, es comentar las líneas del programa que contengan la sentencia @Override, puesto que, al no ser reconocida por el compilador de Jif, es la generadora de errores de compilación.

4.1.3. Casting entre tipos EditText y View

El framework de Android cuenta con diferentes clases para manejar las interfaces gráficas que presenta al usuario, entre las cuales se encuentran EditText y View.
View es la clase principal para la creación de widgets, necesarios para la implementación de componentes interactivos en las interfaces de usuario(UI).
EditText permite adicionar campos de texto editables en UI.
El casting entre los tipos de datos que representan ambas clases, se hace cuando la aplicación debe procesar datos provenientes de campos en las interfaces del usuario, por ejemplo como se observa a continuación:

```
EditText editPassword = (EditText)findViewById(R.id.password);
String password = editPassword.getText().toString();
```

la interfaz de usuario(que es de tipo View) contiene un campo R.id.password, y para manipular la información que almacena, debe ser de tipo EditText, siendo necesario un casting de tipo View a tipo EditText. La dificultad que se presenta con este tipo de casting es que para el sistema de anotaciones de jif no es válido. Luego de probar

con la anotación manual de ambas clases, tratando de dar soporte a este tipo de casting, sin obtener resultados satisfactorios, se opta por "simular" estos casos, es decir, si el tipo de dato de una variable es de tipo EditText, se crea una variable tipo String con un valor determinado, así se omite el casting y se puede analizar el flujo de información.

4.1.4. Clase nested R

El framework de Android utiliza identificadores para hacer referencia a recursos utilizados por la aplicación, recursos como strings, widgets y layouts, tales identificadores son autogenerados en la clase R.java, allí cada recurso es descrito como una clase individual. Al tratarse de una clase nested, la clase R no puede ser anotada con jif. Esto puede solucionarse implementando una versión Jif generalizada de la clase R, que contenga los recursos utilizados en una aplicación, definidos como variables y no como clases.

4.1.5. Enhanced for loop

Además de soportar la estructura de control for, el lenguaje Java permite el uso de enhanced for, que es utilizado para simplificar la iteración en arrays y colecciones, por ejemplo:

```
for(char c : imei.toCharArray())
obfuscated += c + "_";
```

A diferencia de Java, Jif no soporta el enhanced for.
Debido a que ambas sentencias de control son equivalentes, la solución que se propone para poder analizar flujo de información en los aplicativos Android que contengan dicha estructura de control, es generar el equivalente del programa haciendo uso del for, de este modo se cambia la sintaxis sin afectar la lógica del aplicativo a analizar.

4.1.6. Otras limitaciones

Adicional a las limitaciones descritas anteriormente, para las cuales se propone una solución, se identifica que Jif no soporta la sintaxis utilizada para definir estructuras de datos HashMaps, LinkedList y Sets, que en Java se definen de la siguiente manera:

```
Map<String,String> hashMap = new HashMap<String, String>();
List<String> listData;
Set<String> phoneNumbers = new HashSet<String>();
```

Jif tampoco permite la definición de interfaces como argumentos de un método. El siguiente fragmento de código en una aplicación Android, muestra la definición de una interfaz pasada como parámetro al método setOnClickListener.

```
Button button1= (Button) findViewById(R.id.button1);
button1.setOnClickListener(new View.OnClickListener() {
...
.....}  });
```

En estos casos, la dificultad está en encontrar una sintaxis que permita obtener la versión equivalente del programa que las contenga. A lo que se suma, la falta de documentación disponible para solventar los mismos. En consecuencia, se omite del conjunto de aplicaciones evaluables, aplicaciones Android que requieran en su implementación las estructuras de datos descritas.

4.2. Detalles de implementación

Como se describió anteriormente, la implementación del diseño requiere anotaciones a la API de Android y anotaciones en los aplicativos a analizar.
En el caso de las anotaciones requeridas para la API 3.5.2, su implementación se hace manualmente.
En el caso de las anotaciones en el aplicativo a analizar 3.5.3, para las que se propuso un esquema de anotación automatizable, se implementa un prototipo de anotación en lenguaje Java. En la sección 7.1 de los anexos, se incluye su respectivo diagrama de clases.
El anotador está implementado en lenguaje Java y consta de cinco clases: *Main*, *Source*, *XmlExtract*, *Annotation* y *BufferWriter*.
Todos los pasos de anotación descritos en 3.5.3, son coordinados desde la clase *Main*.
La clase *Source* identifica qué variables source contiene la aplicación que se está analizando, esta clase se apoya en la clase *XmlExtract* para identificar sources definidos en interfaces UI. En este caso particular, para la verificación de la política establecida, la clase *XmlExtract* permite identificar el source EditText, cuando se trata de un campo UI que almacena contraseñas.
Como las interfaces UI se describen mediante XML, es necesario extraer los elementos de la interfaz utilizando las librerías javax.xml.parsers y org.w3c.dom. Con javax.xml.parsers se obtiene el respectivo DOM del XML a analizar. Con org.w3c.dom se recorren los elementos del DOM, elementos como los campos EditText.
Con la clase *Annotation* se identifican y anotan los diferentes tipos de métodos(métodos source, no source y métodos a sobreescribir).
En la clase *BufferWriter* se anotan las variables sources, y finalmente se retorna la versión .jif del aplicativo que se está analizando.
Para la anotación del código se utiliza la librería Java Parser 1.8 que permite generar y visitar el árbol de sintaxis para la estructura del código de cada programa, haciendo posible la adición de los labels de seguridad respectivos.
(Incluir Figura)

4.3. Ambiente y herramientas

Versión de la API de Android
Las anotaciones a las clases de la API Android y las aplicaciones a anotar mediante el prototipo, corresponden a la versión Android 4.2.2(API Level 17).

Versión del compilador de Jif

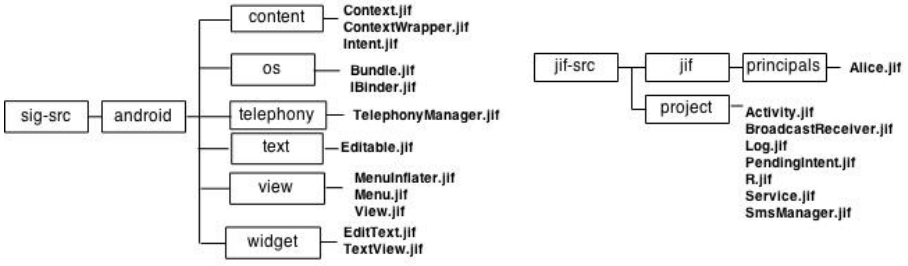

Figura 4.1: Estructura de directorios en Jif. El código con anotaciones que se requiere para la evaluación de flujo de información en Jif, es alojado en los directorios sig-src y jif-src.

El componente que verifica el cumplimiento de las políticas de seguridad, corresponde a la versión 3.4.2 del compilador de Jif.

Estructura de trabajo en JIF
El compilador de Jif maneja una estructura de directorios en la que se incluye el código fuente .jif de las aplicaciones a analizar. Como ilustra la figura 4.1 se tienen los directorios principales sig-src y jif-src. El directorio sig-src está destinado para incluir librerías adicionales.
El directorio jif-src contiene todo el código jif correspondiente al programa como tal que se va evaluar mediante Jif. Para los propósitos del presente trabajo, se tienen los subdirectorios principals y project. En el subdirectorio principals se incluyen las autoridades requeridas, y en el subdirectorio project se incluyen, tanto las clases específicas de los aplicativos Android a analizar, como las clases de que estos heredan como: Activity.jif, Service.jif, etc.

4.4. Compilación, integración y finalmente ejecución

En la sección 7.4 de los anexos, se describen las instrucciones paso a paso de cómo ejecutar el anotador, y cómo compilar los aplicativos a analizar mediante Jif.

CAPÍTULO 5

Evaluación

5.1. Consideraciones de evaluación

El flujo de información es analizado al interior de una sola aplicación, no se consideran flujos de información vía interApp, es decir, varias aplicaciones que se comunican entre sí.

Cabe anotar que los resultados de evaluación que se presentan a continuación son los obtenidos en el siguiente ambiente de pruebas: una máquina virtual con sistema operativo Linux; 2,7 GHz de procesador y 1GB de Memoria RAM. En esa misma máquina, se tiene instalada la versión 3.4.2 del compilador de Jif; el artefacto de prueba para FlowDroid y JoDroid, obtenidos desde, [23] y [24], respectivamente.

5.2. Conjunto de evaluación

Para la evaluación se parte de DroidBech versión 1.0[18], benchmark integrado por casos de prueba para aplicaciones Android, cuyos autores son los mismos creadores de FlowDroid. Se opta por utilizar DroidBench puesto que, en la literatura científica consultada al respecto, no se encuentran otros benchmarks diseñados específicamente para evaluar aplicaciones Android.
De DroidBech se toma un grupo de testcases evaluables frente a la política de seguridad establecida 3.3, este grupo está integrado por 20 testcases. De los cuales, 14 presentan fugas de información.

La tabla 5.1 describe parte del grupo de testcases a evaluar. En los casos en que se requiere, se precisan observaciones entre los resultados de evaluación esperados para la técnica de análisis utilizada por FlowDroid(análisis de flujo de datos) y la técnica de análisis propuesta en el presente trabajo(análisis de flujo de información). En la sección 7.2 de los anexos, se encuentra la descripción del grupo de prueba completo.

El conjunto de prueba es analizado con FlowDroid, JoDroid y con el Prototipo.
Los fundamentos[25] para calificar los resultados del análisis son los siguientes:
True Positive(TP) y False Positive(FP), para referenciar el número de Casos Positivos esperados que son correcta o incorrectatamente identificados.

AndroidSpecific_DirectLeak1	
Descripción	**Leaks**
La variable *mrg* tiene un nivel de seguridad alto, almacena información retornada por el método source *getDeviceId*. Se genera flujo de información directo entre información con nivel de seguridad alto e información con nivel de seguridad bajo, al enviar como parámetro del método *sendTextMessage*, información de la variable *mrg*.	1
AndroidSpecific_LogNoLeak	
Descripción	**Leaks**
El caso de prueba no presenta información con niveles de seguridad alto. Se presentan flujos de información entre información con el mismo nivel de seguridad, en este caso bajo, lo cual es permitido.	0
ArraysAndLists_ArrayAccess1	
Descripción	**Leaks**
Se tiene un array en que se almacena información tanto proveniente como no proveniente de sources, parte de la información que almacena es enviada como parámetro del método *sendTextMessage*. *Observación:* Para la técnica de análisis de FlowDroid(taint analysis), se marca únicamente el índice del array donde se almacena el dato considerado como source, así, cuando se envía como parámetro del método *sendTextMessage*, el dato de un índice no marcado, no se genera leak. Para nuestra técnica de análisis(flujo de información mediante JIF), para que un array almacene información con nivel de seguridad alto, primero debe ser catalogo(anotado) con nivel de seguridad alto, lo que implica que el array podrá almacenar información tanto de nivel de seguridad alto como bajo, pero toda la información quedará con nivel de seguridad alto. En consecuencia, al enviar cualquier índice del array como parámetro del método *sendTextMessage* se presenta un flujo de información no permitido.	0
ImplicitFlows_ImplicitFlow2	
Descripción	**Leaks**
La variable *userInputPassword* con nivel de seguridad alto, almacena información de un campo EditText tipo textPassword(password suministrado por el usuario). Se generan flujos de información indebidos: al tratar de asignar información a la variable passwordCorrect con nivel de seguridad bajo, a partir de la comparación de información con nivel de seguridad alto(variable textPassword), después, al tratar de mostrar en el *log* información que depende de tal comparación.	1

Tabla 5.1: Aplicaciones de prueba.
Describe parte del conjunto de aplicaciones de prueba.

True Negatives(TN) y False Negatives(FN), para referenciar el número de Casos Negativos esperados que son correcta o incorrectamente identificados.
Ahora, para el contexto del presente análisis, donde se evalúa la detección de fugas de información, los resultados del análisis reportados por cada herramienta son calificados como:
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.

Una vez se tienen los resultados de análisis reportados por cada herramienta, se calcula la Precisión y el Recall para cada una de ellas.

La **Precisión** hace referencia a los Casos Positivos esperados(correctos e incorrectos: TP, FP), en contraste con la proporción de verdaderos Positivos(TP) detectados[25]. 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[25]. 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:

$$p = TP/(TP + FP) \tag{5.1}$$

$$r = TP/(TP + FN) \tag{5.2}$$

Donde:
TP representa el total de verdaderos positivos; FP corresponde al total de falsos positivos y FN representa el total de falsos negativos; reportados por la herramienta.

Además de las fórmulas anteriormente descritas, se utiliza el comando *time*[26] de unix, para medir el desempeño de cada herramienta.

5.3. Evaluación FlowDroid y Prototipo

Del total de testcases(20), 14 presentan fugas de información mientras que 6 de ellos no. Los resultados de evaluación para FlowDroid y el Prototipo, son presentados en la tabla 5.2. En esta, por cada caso de prueba se indica la cantidad de leaks que presenta, el resultado devuelto por la herramienta y el tiempo que tarda el análisis.

Item	Testcase	Leaks	F	P	tF	tP
1	AndroidSpecific_DirectLeak1	1	TP	TP	5.371s	2.063s
2	AndroidSpecific_InactiveActivity	0	TN	FP	3.255s	2.469s
3	AndroidSpecific_LogNoLeak	0	TN	TN	5.505s	2.946s
4	AndroidSpecific_Obfuscation1	1	TP	TP	6.734s	2.706s
5	AndroidSpecific_PrivateDataLeak2	1	TP	TP	6.144s	2.644s
6	ArraysAndLists_ArrayAccess1	0	FP	FP	4.708s	1.278s
7	ArraysAndLists_ArrayAccess2	0	FP	FP	4.4s	1.361s
8	GeneralJava_Exceptions1	1	TP	TP	6.397s	2.755s
9	GeneralJava_Exceptions2	1	TP	TP	5.887s	1.980s
10	GeneralJava_Exceptions3	0	FP	FP	6.008s	2.032s
11	GeneralJava_Exceptions4	1	TP	TP	5.731s	2.313s
12	GeneralJava_Loop1	1	TP	TP	5.605s	2.800s
13	GeneralJava_Loop2	1	TP	TP	4.719s	1.361s
14	GeneralJava_UnreachableCode	0	TN	FP	3.792s	1.197s
15	ImplicitFlows_ImplicitFlow1	1	FN	TP	4.853s	1.331s
16	ImplicitFlows_ImplicitFlow2	1	FN	TP	4.496s	1.212s
17	ImplicitFlows_ImplicitFlow4	1	FN	TP	4.375s	1.224s
18	Lifecycle_ActivityLifecycle3	1	TP	TP	4.792s	1.222s
19	Lifecycle_BroadcastReceiverLifecycle1	1	TP	TP	4.456s	1.061s
20	Lifecycle_ServiceLifecycle1	1	TP	TP	5.225s	1.180s

Tabla 5.2: Resultados de evaluación para FlowDroid y Prototipo. Donde *Item* indica el testcase que se evalúa, *Testcase* especifica el nombre de la aplicación para el caso de prueba; *Leaks* indica si el testcase presenta fugas de información; *F* y *P* muestran los resultados devueltos por FlowDroid y por el Prototipo; *tF* y *tP*, señalan el tiempo(en segundos) que toma el análisis para Flowdroid y para el Prototipo, respectivamente.

5.3.1. Análisis de evaluación entre FlowDroid y Prototipo

Resultados de desempeño

Acorde a los tiempos señalados en los campos tF y tP de la tabla 5.2, en promedio, FlowDroid tarda 3,3 segundos más que el Prototipo para ejecutar el análisis.

Falsos positivos, precisiones

En lo que respecta a los resultados del Prototipo, los FP correspondientes a los items 2 y 14 de la tabla 5.2, surgen como consecuencia de un diseño de análisis pesimista (evaluación del flujo de información 3.4), donde se asume que todos los métodos definidos en la clase serán invocados. Por consiguiente, todos los métodos son incluidos para el análisis del flujo de información. Así, si los métodos conllevan a flujos de datos indebidos, independientemente de si son invocados o no, son considerados como generadores de fugas de información.
Por otro lado, en el caso de ArraysAndLists: items 6 y 7, de la tabla 5.2, no es sencillo calificar los resultados como FP, puesto que, para lo que está analizando FlowDroid(verificar que su técnica de análisis diferencie entre los elementos marcados y no marcados de un array), efectivamente se presentan FP, sin embargo, para

la forma en que se deben implementar los programas en Jif, donde se suele definir un nivel de seguridad para todo el array antes de almacenar los elementos en el mismo, podría decirse que no se trata de un FP, porque se revelo información que había sido catalogada con nivel de seguridad alto.

Detección de flujos implícitos

A diferencia de FlowDroid, el Prototipo detecta fugas de información través de flujos implícitos. La no detección de Flujos implícitos por parte de FlowDroid, responde al tipo de análisis y las técnicas en que se fundamenta la herramienta: análisis de flujo de datos mediante técnicas tainting. Basada en asociar una o más marcas con el valor de los datos en el programa, y en propagarlas. Dependiendo de los criterios definidos para el análisis, la marca puede ser propagada a causa de flujos explícitos o de flujos implícitos[11], o a causa de ambos. En flujos explícitos la propagación ocurre cuando el valor de una variable marcada está implicada en el calculo de otra variable. En flujos implícitos la propagación tiene lugar a través de dependencias en el control de flujo del programa, por ejemplo, cuando el valor de un dato marcado afecta indirectamente otra variable.
En el caso de FlowDroid, los criterios que fundamentan el análisis de la herramienta, hacen que el marcado de datos se propague para flujos explícitos y y no para flujos implícitos. Por consiguiente, FlowDroid no detecta flujos implícitos.
Con esto presente, se definen dos escenarios de análisis: Precisión y Recall, incluyendo flujos implícitos; y Precisión y Recall, excluyendo flujos implícitos, donde se incluyen u omiten los testcases para flujos implícitos(items 15 a 17, tabla 5.2).
La tabla 5.3 muestra los resultados de evaluación para ambos casos.

	Con FI		Sin FI	
	FlowDroid	Prototipo	FlowDroid	Prototipo
FP	3	5	3	5
FN	3	0	0	0
TP	11	14	11	11
TN	3	1	3	1

Tabla 5.3: Resultados de precisión para FlowDroid y Prototipo, de acuerdo al escenario, incluyendo o excluyendo flujos implícitos(FI). Resume el total de falsos positivos(FP), verdaderos positivos(TP), verdaderos negativos(TN) y falsos negativos(FN); obtenidos tanto con FlowDroid como con el Prototipo.

Precisión y Recall, incluyendo flujos implícitos

Los resultados obtenidos en la tabla 5.3 señalan que de las 14 fugas existentes, el Prototipo las detecta todas, presenta 14 TP(verdaderos positivos); mientras que, FlowDroid deja pasar 3.
Por otro lado, el Prototipo presenta más falsos positivos que FlowDroid, de los 6 testcases que no presentan leaks, el prototipo reporta 5 como si fuesen fugas, mientras que FlowDroid reporta 3.

Item	Testcase	Leaks	J	P	tJ	tP
1	AndroidSpecific_DirectLeak1	1	TP	TP	22m11.991s	2.063s
2	AndroidSpecific_InactiveActivity	0	FP	FP	22m25.617s	2.469s
3	AndroidSpecific_LogNoLeak	0	TN	TN	21m6.548s	2.946s
4	AndroidSpecific_Obfuscation1	1	TP	TP	22m46.541s	2.706s
5	AndroidSpecific_PrivateDataLeak2	1	TP	TP	21m32.447s	2.644s
6	ArraysAndLists_ArrayAccess1	0	FP	FP	22m01.926s	1.278s
7	ArraysAndLists_ArrayAccess2	0	FP	FP	22m11.023s	1.361s
8	GeneralJava_Exceptions1	1	FN	TP	22m52.134s	2.755s
9	GeneralJava_Exceptions2	1	FN	TP	21m4.434s	1.980s
10	GeneralJava_Exceptions3	0	TN[1]	FP	21m37.040s	2.032s
11	GeneralJava_Exceptions4	1	FN	TP	21m10.240s	2.313s
12	GeneralJava_Loop1	1	TP	TP	21m15.30s	2.800s
13	GeneralJava_Loop2	1	TP	TP	21m41.224s	1.361s
14	GeneralJava_UnreachableCode	0	TN	FP	22m84.138s	1.197s
15	ImplicitFlows_ImplicitFlow1	1	TP	TP	22m55.645s	1.331s
16	ImplicitFlows_ImplicitFlow2	1	TP	TP	22m32.231s	1.212s
17	ImplicitFlows_ImplicitFlow4	1	TP	TP	22m43.110s	1.224s
18	Lifecycle_ActivityLifecycle3	1	TP	TP	22m54.651s	1.222s
19	Lifecycle_BroadcastReceiverLifecycle1	1	TP	TP	22m42.347s	1.061s
20	Lifecycle_ServiceLifecycle1	1	TP	TP	22m92.722s	1.180s

Tabla 5.4: Resultados de evaluación para JoDroid y Prototipo. Donde *Testcase* especifica el nombre de la aplicación que se está evaluando; *Leaks* indica si el testcase presenta fugas de información; J y P muestran los resultados devueltos por JoDroid y por el Prototipo; tJ y tP, señalan el tiempo que toma el análisis para JoDroid y para el Prototipo, respectivamente.

Así, en lo que respecta a Precisión, FlowDroid presenta un porcentaje del 78,57 %, siendo más preciso frente al Prototipo, que presenta un porcentaje del 73,68 %.
Por otro lado, el Prototipo presenta un porcentaje en Recall del 100 %, mientras que FlowDroid presenta un porcentaje del 78,75 %.

Precisión y Recall, excluyendo flujos implícitos

Excluyendo los testcases para flujos implícitos, el conjunto de prueba se reduce a 17 casos. De los cuales, 11 presentan leaks.
En este contexto, el porcentaje de Precisión para FlowDroid es de 78,57 %, mientras que para el Prototipo es del 68,75 %. El porcentaje de Recall es igual para FlowDroid y el Prototipo: 100 %.

5.4. Evaluación JoDroid y Prototipo

La tabla 5.4 ilustra los resultados de evaluación para JoDroid y el Prototipo, en base a los cuales, se calculan las métricas de precisión y recall.

[1] Al igual que en el resto de testcases para GeneralJava_Exceptions(items 8, 9 y 11), la herramienta no detecta leaks, la diferencia para el presente caso, es que efectivamente no existe leak. Por tanto se califica como TN.

	Con excepciones		Sin excepciones	
	JoDroid	Prototipo	JoDroid	Prototipo
FP	3	5	3	4
FN	3	0	0	0
TP	11	14	11	11
TN	3	1	2	1

Tabla 5.5: Resultados de precisión para JoDroid y Prototipo. Muestra los escenarios en que mide. Resume el total de falsos positivos(FP), verdaderos positivos(TP), verdaderos negativos(TN) y falsos negativos(FN); obtenidos tanto con JoDroid como con el Prototipo.

5.4.1. Análisis de evaluación entre JoDroid y Prototipo

Resultados de desempeño

Para el análisis mediante JoDroid se deben seguir una serie de pasos, tal como se describen en el manual de referencia[24], de estos, únicamente se toma el tiempo correspondiente al paso para generar el grafo de dependencia del programa(PDG), del cual parte el análisis. En general, el tiempo que tarda la generación del PDG para cada aplicación analizada, oscila entre 21 y 22 minutos. Cabe anotar que estos valores podrían cambiar con otras características de hardware, sin embargo, asignando 1 GB de Ram a la máquina virtual de Java, para la generación del PDG, es ese el rango de tiempo obtenido. En consecuencia, para los valores de tiempo que señala la presente evaluación, es posible decir que la herramienta es costosa en desempeño.

Detección de Flujos implícitos

La tabla 5.4 muestra que al igual que el Prototipo, JoDroid detecta fugas de información a través de flujos implícitos. Ya que, en los testcases correspondientes a flujos implícitos, items 15, 16 y 17, ambas herramientas detectan que efectivamente, se presentan fugas de información.

Fugas a través de excepciones

Es importante resaltar que del conjunto de casos de prueba, JoDroid ignora el control de flujo de información para excepciones(items 8 a 11 de la tabla 5.4), puesto que, su actual implementación no soporta análisis del flujo de información a través de tales sentencias[pag 93 [27]]. En la tabla 5.5 se muestran dos escenarios para los resultados de evaluación: Con excepciones y Sin excepciones. Con base en tales resultados se calcula la Precisión y Recall para cada uno de los escenarios.

Precisión y Recall incluyendo excepciones

Del total de testcases(20), 14 presentan fugas de información. De los casos con fuga de información, 3 corresponden a las excepciones incluidas(items 8 a 11, tabla 5.4), y se califican como falsos negativos(FN) porque JoDroid no los detecta. La tabla 5.5 ilustra los resultados de evaluación.
En cuanto a la Precisión(p), JoDroid presenta un porcentaje del 78,57 %, mientras que el Prototipo presenta un porcentaje del 73,68 %.
En cuanto a Recall(r), el Prototipo presenta un porcentaje del 100 %, frente a un porcentaje del 78,57 % presentado por JoDroid.

Precisión y Recall excluyendo excepciones

Omitiendo los testcases para excepciones(items 8 a 11, tabla 5.4), el total de testcases(20) queda reducido a 16. De estos, 11 presentan fugas de información.
En lo que respecta a la métrica de Precisión, JoDroid presenta un porcentaje del 78,57 %; frente al Prototipo que presenta un porcentaje del 73,33 %.
Para la métrica de Recall, tanto JoiDroid como el Prototipo, presentan el mismo porcentaje esto es 100 %.

5.5. Análisis de evaluación FlowDroid, JoDroid, Prototipo

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

Tabla 5.6: Resultados de precisión para FlowDroid y Prototipo. Resume el total de falsos positivos(FP), verdaderos positivos(TP), verdaderos negativos(TN) y falsos negativos(FN).

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

Tabla 5.7: Comparación entre FlowDroid, JoDroid y Prototipo. Ilustra los porcentajes para Precisión, Recall, y la detección de leaks mediante flujos implícitos.

En base a los resultados para el conjunto de evaluación(compuesto por 20 testcases, de los cuales 14 presentan leaks), obtenidos en los anteriores apartados, se comparan las tres herramientas: FlowDroid, JoDroid y Prototipo, frente a Precisión, Recall, y

la detección de fugas de información mediante flujos implícitos. La tabla 5.6 ilustra todos los resultados y la tabla 5.7 ilustra los respectivos porcentajes.

Desempeño
Como muestran las tablas 5.2 y 5.4, el Prototipo presenta mejor desempeño frente FlowDroid y JoDroid. En el caso de FlowDroid, en promedio tarda 3,3 segundos más que el Prototipo para ejecutar el análisis. En el caso de JoDroid, el tiempo de análisis es costoso en comparación a las otras herramientas, puesto que su tiempo de ejecución oscila entre 21 y 22 minutos.

Precisión y Recall
Tanto FlowDroid como JoDroid presentan mejor Precisión que el Prototipo, es decir que el Prototipo presenta más falsos positivos(FP).
Por otro lado, el Prototipo presenta mayor Recall frente a FlowDroid y JoDroid, por tanto, el Prototipo detecta mayor cantidad de fugas existentes (reporta menos FN). Para este caso particular, el Prototipo detecta todos los TP.
En consecuencia, es posible decir que aunque el Prototipo presenta mayor cantidad de FP frente a FlowDroid y JoDroid, deja pasar menos fugas de información.
En lo que respecta a flujos implícitos, a diferencia de FlowDroid, tanto JoDroid como el Prototipo detectan fugas de información a través de Flujos implícitos.

Análisis métricas acorde al tipo de análisis
Analizando los resultados para las métricas de desempeño, precisión y recall; descritas anteriormente, acorde al tipo de análisis y técnicas en que se basa cada herramienta, es posible anotar:

- Desempeño:
El prototipo presenta mejor desempeño, como resultado de analizar flujo de información mediante lenguajes tipados de seguridad, más específicamente a través de Jif. Dado que Jif recurre a técnicas de compilación(label checking) y no requiere la generación de grafos de dependencia, el análisis toma menos tiempo.

-Precisión, Recall y detección de Flujos implícitos:
el análisis pesimista en que se basa el Prototipo, donde se asume que todos los métodos implementados en la aplicación serán invocados, hace que los resultados del análisis sean menos precisos, generando más falsos positivos(FP). Para la evaluación realizada, el análisis de flujo de información mediante el sistema de anotaciones de Jif, ofrece un mejor recall, frente a: el 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. Esto hace que para el experimento, la técnica de análisis del prototipo sea completa(completeness), puesto que, dentro de los leaks detectados están todos los leaks que efectivamente existen.
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 la cuales, sino se propaga el marcado de datos a través de flujos implícitos, no es posible detectar fugas de datos a través de

los mismos.

Los resultados de evaluación confirman las hipótesis iniciales del presente trabajo, según las cuales se esperaba que: al hacer análisis de flujo de información mediante lenguajes tipados de seguridad, los resultados del análisis fuesen más rápidos pero menos precisos, reportando más falsos positivos que JoDroid y FlowDroid.
Tal resultado refleja lo ilustrado por la teoría [19] y [28].

La tabla 5.8 resume las ventajas, desventajas, similitudes y diferencias entre el Prototipo y las herramientas de comparación, FlowDroid y JoDroid, respectivamente. En esta se ilustra por ejemplo, como el prototipo detecta automáticamente los sources y sinks, mientras que JoDroid no.

Item	**Prototipo vs FlowDroid**				**Prototipo vs JoDroid**			
	ventaja	**desvent**	**similit**	**diff**	**ventaja**	**desvent**	**similit**	**diff**
Menor Precisión		✓				✓		
Mayor Recall	✓				✓			
Menor costo en desempeño					✓			
Bajo costo en desempeño			✓					
Detección de flujos implícitos	✓						✓	
No detección automática de sources y sinks		✓					✓	
No soporte para Análisis interApp		✓					✓	
Tipo de análisis(flujo de información; flujo de datos)				✓				
Tipo de análisis IFC							✓	
Técnica de análisis: PDG, slicing								✓

Tabla 5.8: Síntesis ventajas, desventajas, similitudes y diferencias; del Prototipo frente a FlowDroid y JoDroid(respectivamente).

5.6. Tipos de análisis y técnicas evaluadas

FlowDroid se fundamenta en análisis de flujo de datos, mediante técnicas tainting. El código .dex a ser analizado es transformado a una representación intermedia(Jimple representation).
El análisis parte de la construcción de un super-grafo del programa que se analiza, el super-grafo es una colección de grafos dirigidos, mediante los cuales se representa el programa, donde los nodos asocian las sentencias del programa y las aristas, la forma en que estas se conectan. Para recorrer el super-grafo utiliza un algoritmo basado en el problema de graph-reachability[29]; cuyo costo computacional es de orden polinomial O(ED3), donde E representa funciones de flujo de datos(dataflow functions) y D conjunto de elementos para guiar el seguimiento de los datos marcados(set of data flow facts).

Para propagar la marca en los datos que analiza omite el control de flujo de información, sólo se centra en el flujo de datos marcados como sources y sinks.
La herramienta recibe como entrada el apk del aplicativo, detecta automáticamente los sources y sinks del programa mediante el uso de SuSi y genera un reporte del análisis.

JoDroid se fundamenta en análisis de control de flujo de información, aplicando técnicas de grafos de dependencia(PDG) y técnicas slicing.
El código .dex es transformado a código de representación intermedio(SSA-form). Construye un grafo PDG, donde los nodos representan statements y expresiones, y las aristas modelan las dependencias sintácticas entre los statements y expresiones. Este PDG permite modelar flujos explícitos e implícitos.
El costo computacional un análisis basado en PDG es de orden polinomial O(N)3[20, page 3].
Para hacer seguimiento al control de flujo de información, utiliza labels de seguridad, estos califican con nivel de seguridad alto o bajo información de variables y statements.
Los procedimientos para usar la herramienta comprenden: generar el punto de entrada del análisis, generar el PDG, ejecutar el respectivo análisis. Primero, recibe como entrada el apk y manifest del aplicativo para generar un archivo con el punto de entrada del análisis; luego, a partir del archivo devuelto anteriormente genera el PDG, finalmente, recibe como entrada el PDG, lista los statements y variables del aplicativo para que se indique manualmente los sources y sinks, y genera el respectivo análisis.

La propuesta está basada en análisis de flujo de información mediante lenguajes tipados de seguridad, más específicamente mediante Jif.
Para cada programa a analizar se debe implementar la versión Jif, es decir, el programa implementado acorde al sistema de anotaciones de Jif. A partir de tales anotaciones el compilador verifica la generación de flujos de información que incumplan la política de seguridad establecida, para reportarlos como flujos de información indebidos. Al ser evaluado directamente por un compilador, obtiene los beneficios de bajo costo computacional del mismo.
La generación del análisis para verificar la política de seguridad definida, requiere dos pasos. Primero, se genera la versión Jif del aplicativo a analizar; Segundo, se compila el .jif, para obtener el reporte de análisis.

En el cuadro 5.9 se resumen los puntos comparados anteriormente.

Herramienta	**Tipo**	**Técnicas**	**Costo computacional**	**Entradas**
FlowDroid	Flujo de datos	Tainting; super-grafo integrado por grafos dirigidos; Representación intermedia Jimple; algoritmo graph-reachability	Polinomial O(ED3)[29]	apk
JoDroid	Flujo de información	PDG; slicing; Representación intermedia(SSA- form)	polinomial O(N)3[20]	apk; Manifest; sources y sinks
Prototipo	Flujo de información	Lenguajes tipados de seguridad; Type checking	Tiempo de compilación(Tiempo realmente bajo)	código fuente

Tabla 5.9: Generalidades técnicas de análisis evaluadas

CAPÍTULO 6

Trabajo Futuro y Conclusiones

6.1. Discusión

Límites de la solución propuesta

Las limitaciones de la solución propuesta se enmarcan en: políticas evaluables y características del lenguaje Jif.
Por un lado, 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.

Por otro lado, están las limitaciones propias del lenguaje Jif, es decir, características para el lenguaje Java estándar que aún no están soportadas por el compilador de Jif, y que por tanto, impiden analizar el flujo de información de aplicativos Android que requieran de tales características del lenguaje Java. Más específicamente, se hace referencia a las limitaciones descritas en 4.1: nested clases, initializer blocks, threads, etc.

6.1.1. Qué tanto cambia el código original con las anotaciones

En el diseño de la solución se describieron los retos técnicos 4.1 que implica anotar código Android, y cómo superar algunos de estos. Específicamente, las limitaciones para las que se propone un mecanismo que permita soportarlas son: Statement @Override; Casting entre tipos EditText y View; Clase Nested R y Enhanced for loop.

Adicionalmente, se describió que el compilador de Jif exige la declaración del chequeo de excepciones tipo runtime, en programas que lo requieran3.4. Ahora, cumplir con los requisitos del compilador de Jif y aplicar mecanismos que soporten tales limitaciones, implica una serie de transformaciones en el código fuente del programa Android a analizar, tanto en la etapa previa a la anotación como en la anotación

misma.
Previa a la anotación el desarrollador debe garantizar dos cosas, primero debe adicionar las runtime exceptions que su programa necesite, segundo cuando requiera el uso de bucles for, debe usar la versión sencilla y no la versión mejorada del mismo(Enhanced for loop).
Durante la anotación, además de aplicar los criterios de anotación definidos en el diseño de la solución 3.5.3, se aplican los mecanismos propuestos para soportar el Statement @Override, Casting entre tipos EditText y View; y Clase Nested R. Así, en el caso de la sentencia Statement @Override, se comenta la línea que lo contenga; para el casting entre tipos EditText y View, la información implicada en este tipo de casting es abstraída mediante un tipo de dato String; finalmente para la clase nested R se crea una clase que define los define los recursos utilizados por la aplicación a través de variables.
Si bien, la idea que fundamenta el diseño de la solución consiste en generar la versión Jif del aplicativo Android a analizar, lo cual se traduce en adicionar las anotaciones correspondientes para evaluar determinada política de seguridad, de modo que el compilador de Jif entienda el programa y permita analizar el flujo de información en el mismo; no es suficiente con sólo anotar el código, en otras palabras, sin los ajustes previamente mencionados, el compilador genera error. Por otro lado, lo positivo es que tales ajustes no alteran la lógica del programa.

6.2. Trabajo Futuro

Cómo puede ser extendido el trabajo y qué beneficios tendría esa extensión

Exceptuando las características del lenguaje Java que no son soportadas por el compilador de Jif(nested clases, initializer blocks, threads)**??**, se podría ampliar el setup de Jif para clases de la API de Android, de modo que se brinde soporte mediante el sistema de anotaciones de Jif a la mayor cantidad de clases posibles de la API. Esto permitiría hacer análisis de flujo de información a aplicaciones Android más robustas.
El esquema de anotación propuesto podría ser extendido para definir y analizar políticas de integridad y mecanismos adicionales como declasificación y endorsment, 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.

6.3. Conclusiones

Qué aprendimos con este trabajo.

Durante el presente trabajo de investigación se ha abordado la problemática que enfrenta 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 acceso de los usuarios a determinados recursos del sistema, y no en verificar qué sucede con la información una vez se accede a ella.

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 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 confidencialidad 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, es pertinente anotar que:
- Con el sistema de anotaciones de Jif es posible proveer una herramienta para que el desarrollador Android 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 ofrece la ventaja de detección de flujos implícitos.
- Al tratarse de análisis de flujo de información mediante lenguajes tipados de seguridad, se tienen las ventajas de desempeño y completitud en el análisis(se presentan menos falsos negativos), pero al mismo tiempo, se obtiene baja precisión(reporte de falsos positivos).

Sistema de anotaciones de Jif y sus retos
Extender el setup de Jif para Android implica importantes retos técnicos, porque además de requerirse la implementación de las clases de la API de Android, mediante el sistema de anotaciones de Jif, es necesario extender características del lenguaje Java estándar, por ejemplo, es necesario dar soporte a las clases java.lang.Override, tal como se describe en 4.1.2.
Adicionalmente, para aspectos del código que definitivamente no puedan ser soportadas mediante el sistema de anotaciones de Jif, se debe optar por un mecanismo que permita soportarlas por ejemplo, como se menciona en 6.1.1.

Por último, aunque el análisis de aplicativos Android mediante el sistema de anotaciones de Jif es un tema de investigación con bastantes retos por superar, el presente trabajo evidencia que es posible anotar código de la API Android mediante Jif, de modo que el desarrollador construya aplicativos que cumplan con determinadas propiedades de seguridad.

CAPÍTULO 7

Anexos

7.1. Diagrama de clases para el anotador

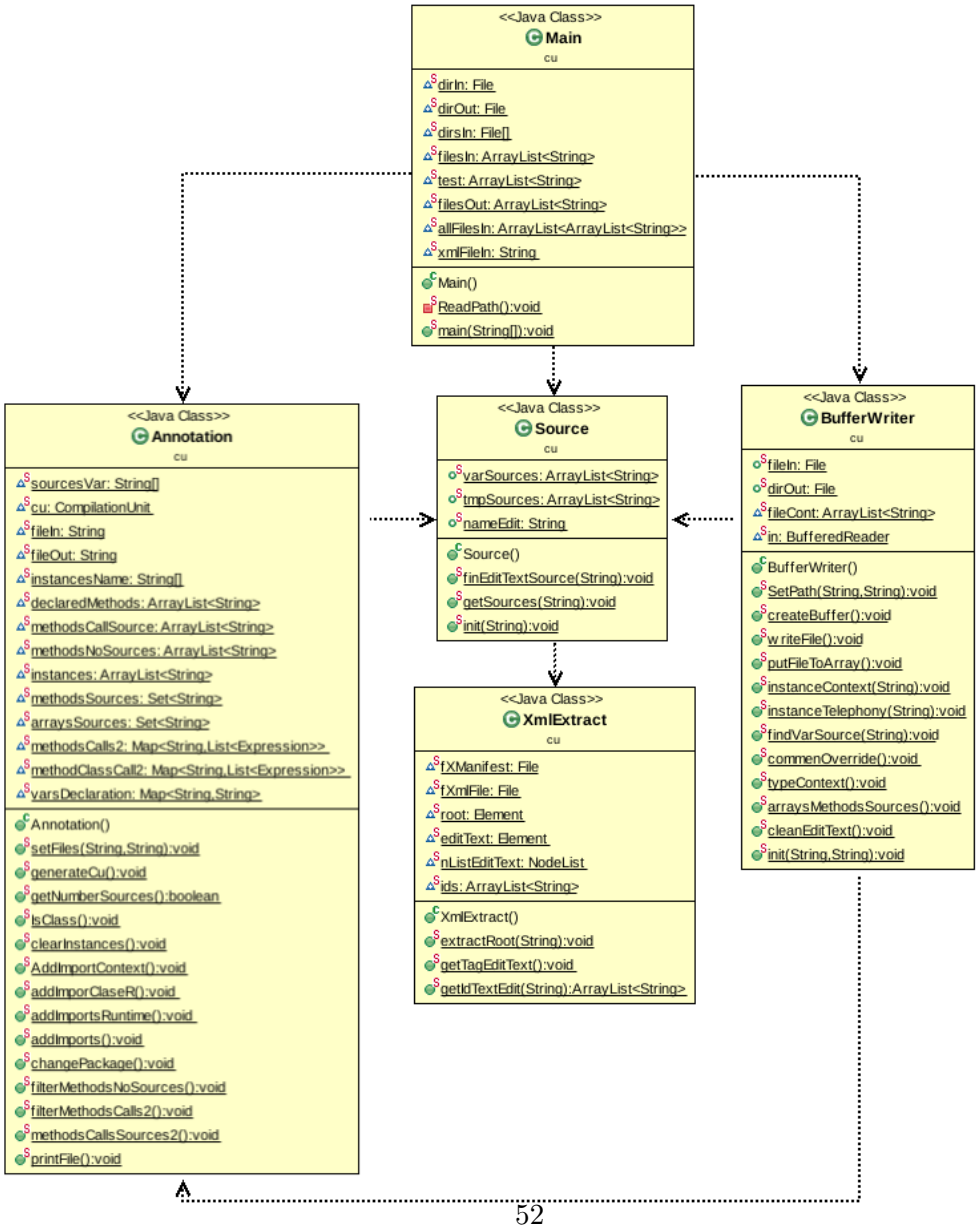

Figura 7.1: Clases necesarias para la implementación del anotador

7.2. Descripción de testcases para evaluación

En las tablas 7.1, 7.2 y 7.3, se describe el comportamiento de los casos de prueba evaluados, donde:
Se considera con nivel de seguridad alto, variables y métodos que almacenan y modifican(respectivamente), información catalogada como privada(Sources).
Se considera con nivel de seguridad bajo, canales para envío de mensajes, muestra de logs y canales creados durante el control de flujo del programa.

En los casos en que se requiere, se precisan observaciones entre los resultados de evaluación esperados para la técnica de análisis utilizada por FlowDroid y la técnica de análisis propuesta en el presente trabajo.

Tabla 7.1: Descripción aplicaciones de prueba

AndroidSpecific_DirectLeak1	
Descripción	**Leaks**
La variable *mrg* tiene un nivel de seguridad alto, almacena información retornada por el método source *getDeviceId*. Se genera flujo de información directo entre información con nivel de seguridad alto e información con nivel de seguridad bajo, al enviar como parámetro del método *sendTextMessage*, información de la variable *mrg*.	1
AndroidSpecific_InactiveActivity	
Descripción	**Leaks**
La variable *imei* tiene un nivel de seguridad alto, almacena información retornada por el source getDeviceId. La variable es enviada como parámetro a *Log*, canal que muestra información con nivel de seguridad bajo. *Observación:* debido a que la actividad en que se presenta este flujo de información no está activada en el Manifest de la aplicación, para la técnica de análisis de FlowDroid no existen leaks. Para nuestra propuesta de análisis si existe leak, porque se asume que los métodos y sus aplicaciones podrán ser ejecutados.	0
AndroidSpecific_LogNoLeak	
Descripción	**Leaks**
El caso de prueba no presenta información con niveles de seguridad alto. Se presentan flujos de información entre información con el mismo nivel de seguridad, en este caso bajo, lo cual es permitido.	0
AndroidSpecific_Obfuscation1	
Descripción	**Leaks**
La variable ***mrg*** tiene un nivel de seguridad alto, almacena información retornada por el método source getDeviceId(). Se genera flujo de información entre información con nivel de seguridad alto e información con nivel de seguridad bajo, al enviar como parámetro del método *sendTextMessage*, información de la variable *mrg*. *Observación:* el elemento adicional para este testcase es proveer una suplantación de la clase android.telephony.TelephonyManager, en el apk de la aplicación. Para la evaluación que proponemos, se verifica acorde a la versión que se tiene anotada para esta clase, es decir, independientemente de la ofuscación de la clase, nuestro análisis debe detectar que existe un flujo de información indebido.	1
AndroidSpecific_PrivateDataLeak2	
Descripción	**Leaks**
La variable *info* tiene un nivel de seguridad alto, almacena información suministrada por el campo EditText de tipo textPassword. Se genera flujo de información entre información con nivel de seguridad alto e información con nivel de seguridad bajo, al pasar la variable *info* como parámetro de *Log*, que muestra información con nivel de seguridad bajo.	1
ArraysAndLists_ArrayAccess1	
Descripción	**Leaks**
Se tiene un array en que se almacena información tanto proveniente como no proveniente de sources, parte de la información que almacena es enviada como parámetro del método *sendTextMessage*. *Observación:* Para la técnica de análisis de FlowDroid(taint analysis), se marca únicamente el índice del array donde se almacena el dato considerado como source, así, cuando se envía como parámetro del método *sendTextMessage*, el dato de un índice no marcado, no se genera leak. Para nuestra técnica de análisis(flujo de información mediante JIF), para que un array almacene información con nivel de seguridad alto, primero debe ser catalogo(anotado) con nivel de seguridad alto, lo que implica que el array podrá almacenar información tanto de nivel de seguridad alto como bajo, pero toda la información quedará con nivel de seguridad alto. En consecuencia, al enviar cualquier índice del array como parámetro del método *sendTextMessage* se presenta un flujo de información no permitido.	0

Tabla 7.2: Descripción aplicaciones de prueba

ArraysAndLists_ArrayAccess2	
Descripción	**Leaks**
Se presenta el contexto descrito en ArraysAndLists_ArrayAccess1, con un elemento adicional, se implementa el método calculateIndex(), que calcula el índice del array a ser enviado como parámetro del método *sendTextMessage*.	0
GeneralJava_Exceptions1	
Descripción	**Leaks**
La variable *imei* es de nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*. Se genera flujo de información entre información de nivel de seguridad alto e información con nivel de seguridad bajo, al enviar como parámetro del método *sendTextMessage* información de la variable *imei*. Este flujo de información se presenta dentro de la captura de una excepción RuntimeException(no es verificada en tiempo de compilación).	1
GeneralJava_Exceptions2	
Descripción	**Leaks**
La variable *imei* es de nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*. El control de flujo del programa conduce de manera implícita a la captura de una excepción tipo RuntimeException, desde allí se utiliza información proveída por la variable *imei*, como parámetro para invocar el método *sendTextMessage*. Generando un flujo de información indebido.	1
GeneralJava_Exceptions3	
Descripción	**Leaks**
La variable *imei* es de nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*. La información proveída por *imei* es utilizada como parámetro para invocar el método *sendTextMessage* dentro de la captura de una excepción tipo RuntimeException, sin embargo, el programa no genera un caso que haga ejecutar la captura de la excepción.	0
GeneralJava_Exceptions4	
Descripción	**Leaks**
La variable *imei* es de nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*. información proveída por esta variable es enviada como parámetro para la captura de una excepción en tiempo de ejecución, donde es utilizado como parámetro para invocar el método *sendTextMessage*, generando un flujo de información indebido.	1
GeneralJava_Loop1	
Descripción	**Leaks**
La variable *imei* es de nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*. Se generan flujos de información indebidos, primero al tratar de asignar la información de la variable a un array con nivel de seguridad bajo(donde se intenta ofuscar la información), luego al tratar de enviar la información ofuscada como parámetro del método *sendTextMessage*, con nivel de seguridad bajo.	1
GeneralJava_Loop2	
Descripción	**Leaks**
La variable *imei* es de nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*. Se busca ofuscar la información de *imei* mediante ciclos for anidados, allí se asigna la información de la variable a un array con nivel de seguridad bajo. Luego se envía la información ofuscada, como parámetro del método *sendTextMessage*, con nivel de seguridad bajo, generando otro flujo de información indebido.	1

Tabla 7.3: Descripción aplicaciones de prueba

GeneralJava_UnreachableCode	
Descripción	**Leaks**
La variable *deviceid* con nivel de seguridad alto, está contenida en un método que no es llamado, dentro del mismo, *deviceid* es pasada como parámetro para invocar el método *sendTextMessage*, cuyo nivel de seguridad es bajo. *Observaciones:* para el análisis de FlowDroid el programa no presenta leaks, ya que el método nunca es llamado. Para nuestro análisis, el programa presenta leak porque se asume que todos los métodos son llamados.	0
ImplicitFlows_ImplicitFlow1	
Descripción	**Leaks**
La variable *imei* con nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*, *imei* se pasa como parámetro al método obfuscateIMEI que devuelve la información ofuscada. Después se invoca el método WriteToLog, con la información ofuscada como parámetro para ser mostrada en el log. Al invocar el método WriteToLog con la información ofuscada, se genera un flujo de información indebido.	1
ImplicitFlows_ImplicitFlow2	
Descripción	**Leaks**
La variable *userInputPassword* con nivel de seguridad alto, almacena información de un campo EditText tipo textPassword(password suministrado por el usuario). Se generan flujos de información indebidos: al tratar de asignar información a la variable passwordCorrect con nivel de seguridad bajo, a partir de la comparación de información con nivel de seguridad alto(variable textPassword), después, al tratar de mostrar en el *log* información que depende de tal comparación.	1
ImplicitFlows_ImplicitFlow4	
Descripción	**Leaks**
La variable *password* con nivel de seguridad alto, almacena información de un campo EditText tipo textPassword, *password* es utilizada como parte de los parámetros para invocar el método *lookup* que busca identificar el password suministrado por el usuario. Se genera un flujo de información indebido, cuando se compara lo retornado por el método para mostrar en el *log* información del password.	1
Lifecycle_ActivityLifecycle3	
Descripción	**Leaks**
El flujo de información entre información con nivel de seguridad alto e información con nivel de seguridad bajo, tiene lugar a través de dos métodos del ciclo de vida de la actividad: onSaveInstanceState y onRestoreInstanceState. En onSaveInstanceState, se asigna información con nivel de seguridad alto a la variable *s*, la información que almacene este método es utilizada durante la reanudación de la actividad, a través del método onRestoreInstanceState, donde se muestra en el *log* información de la variable *s*.	1
Lifecycle_BroadcastReceiverLifecycle1	
Descripción	**Leaks**
Se tiene un broadcast receiver que muestra información con nivel de seguridad alto, contenida en la variable *imei*(almacena información retornada por el método *getDeviceId*) a través del *log*.	1
Lifecycle_ServiceLifecycle1	
Descripción	**Leaks**
Se tiene un servicio que presenta flujo de información indebida mediante dos métodos de su ciclo de vida. En el método que inicia el servicio onStartCommand, la variable con nivel de seguridad alto, almacena información devuelta por el método *getDeviceId*. Luego el método onLowMemory, se envía información de la variable *secret* a través de un mensaje msm.	1

7.3. Formulas

Las formulas 7.1 y 7.2 son aplicadas para calcular los porcentajes de Precisión(p) y Recall(r).

$$p = TP/(TP + FP) \tag{7.1}$$

$$r = TP/(TP + FN) \tag{7.2}$$

Donde:
TP representa el total de verdaderos positivos; FP corresponde al total de falsos positivos y FN representa el total de falsos negativos; reportados por la herramienta.

7.4. Instrucciones para probar el prototipo

En el directorio `/home/testing/eule` están los elementos necesarios para evaluar los casos de prueba, de allí interesan los subdirectorios `androidFlows`, `InputLabelGenerator` y el jar `LabelGenerator.jar`.

El subdirectorio `androidFlows` contiene la estructura de archivos necesaria para ejecutar un programa jif, así: `sig-src` aloja clases java y clases de la API de Android, con signaturas para que jif las reconozca de forma nativa. `jif-src/test` tiene clases de la API de Android con anotaciones jif(Activity.jif, BroadcastReceiver.jif, Log.jif, R.jif, Service.jif, SmsManager.jif). Allí se deben alojar los programas jif a ejecutar.
En `InputLabelGenerator` están los fuentes java a pasar como entrada para el generador de labels(LabelGenerator.jar), que devuelve la versión jif de los mismos. Se recomienda utilizar estos, ya que contienen las adaptaciones necesarias para poder ser analizadas con JIF, la adición de excepciones NullPointerException, ClassCastException y ArrayIndexOutOfBoundsException, son algunos ejemplos de elementos adicionados.

Instrucciones de ejecución:
(1) Ejecutar el jar para la generación de los labels:

```
testing@debianJessie:~/eule$ java -jar LabelGenerator.jar
```

Una vez se ejecuta el .jar, se solicita el directorio de entrada(que contiene las aplicaciones a anotar) y el directorio de salida(para alojar las aplicaciones anotadas). Separados por el simbolo @

```
Ingrese la ruta completa para el directorio de entrada, y para el
directorio de salida:
Ejemplo: dir-entrada@dir-salida
```

Se deben pasar los directorios:

```
InputLabelGenerator@androidFlows/jif-src/test/
```

(2) ejecutar el script setup.sh(basta con ejecutarlo una sola vez)

```
testing@debianJessie:~/eule/androidFlows$ ./setup.sh
```

(3) Ejecutar el .jif generado:
En la ruta pasada como directorio de salida en el punto anterior `androidFlows/jif-src/test`, se genera un subdirectorio por aplicación, con un .java y un *-out.jif. Se debe ejecutar el *-out.jif. Por ejemplo, para evaluar el testcase ArraysAndLists_ArrayAccess1:

```
testing@debianJessie:~/eule/androidFlows$ ./jifc-java-libraries.sh \
jif-src/test/ArraysAndLists_ArrayAccess1/ArrayAccess1-out.jif
```

Cuando se presentan flujos indebidos, el compilador genera una salida señalando los problemas de seguridad.

```
estudiante@debianJessie:~/eule/androidFlows$ ./jifc-java-libraries.sh \
jif-src/test/ArraysAndLists_ArrayAccess1/ArrayAccess1-out.jif
/home/testing/eule/androidFlows/jif-src/test/ArraysAndLists_ArrayAccess1/ArrayAccess1-out.jif:51:
Unsatisfiable constraint
        general constraint:
                actual_arg_3 <= formal_arg_3
        in this context:
                {Alice->; _<-_      caller_pc} <= {}
        cannot satisfy equation:
                {Alice->}     {}
        in environment:
                {this}     {caller_pc}
                []

    Label Descriptions
    ------------------
     - actual_arg_3 = the label of the 3rd actual argument
     - actual_arg_3 = {Alice->; _<-_     caller_pc}
     - formal_arg_3 = the upper bound of the formal argument text
     - formal_arg_3 = {}
     - caller_pc = The pc at the call site of this method (bounded above by
    {})
     - this = label of the special variable "this" in test.ArrayAccess1

    The label of the actual argument, actual_arg_3, is more restrictive than
    the label of the formal argument, formal_arg_3.
            sms.sendTextMessage("+49_1234", null, arrayData[2], null, null);
                                                  ^-------^

1 error.
testing@debianJessie:~/eule/androidFlows$
```

Cuando el caso de prueba no presenta flujos indebidos, el compilador no genera salidas, por ejemplo, al evaluar el testcase AndroidSpecific_LogNoLeak, el compilador retorna el prompt de la shell, sin ningún comentario.

7.5. Instrucciones para uso de FlowDroid

En el directorio `/home/estudiante/eule` también se encuentran los subdirectorios `/FlowDroid` y `/DroidBench-master` que contienen los elementos necesarios para probar los testcases con FlowDroid.
Para ello se requiere ejecutar el jar de FlowDroid, indicando el apk a analizar, los apk están en el subdirectorio DroidBench-master. Por ejemplo, para analizar el testace ImplicitFlow4:

```
testing@debianJessie:~/eule/FlowDroid$ java -jar FlowDroid.jar \
../DroidBench-master/apk/ImplicitFlows/ImplicitFlow4.apk
/home/estudiante/android-sdks/platforms/
```

El archivo `howRunIt` contenido en el directorio `/FlowDroid`, indica como se debe ejecutar.

Bibliografía

[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 *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] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] L. Haav and P. Laud, "Typing computationally secure information flow in jif," *In proceedings of Nordsec 2008, 13th Nordic Workshop on Secure IT Systems, Lyngby, Denmark,*, October 9-10 2008.

[13] 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.

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

[15] 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.

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

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

[18] Secure Software Engineering Group at EC SPRIDE. (2013, June) Droidbench – benchmarks. [Online]. Available: https://github.com/secure-software-engineering/DroidBench/blob/e64bf483949bf4cb91af642a415fadc9c65e4be5/README.md

[19] M. Taghdiri, G. Snelting, and C. Sinz, "Information flow analysis via path condition refinement," in *7th International Workshop on Formal Aspects in Security and Trust (FAST)*, September 2010, pp. 65–79.

[20] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, Dec. 2009.

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

[22] 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

[23] Secure Software Engineering EC SPRIDE. (2015, May) Pldi'14 artifact evaluation. [Online]. Available: https://github.com/secure-software-engineering/soot-infoflow-android/wiki/PLDI'14-Artifact-Evaluation

[24] Karlsruhe Institute of Technology. (2015, March) https://github.com/jgf/joana/blob/master/wala/joana.wala.jodroid/readme.md. [Online]. Available: https://github.com/jgf/joana/blob/master/wala/joana.wala.jodroid/README.md

[25] 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.

[26] Panagiotis Christias. (2015, April) Unix on-line man pages. [Online]. Available: http://dell9.ma.utexas.edu/cgi-bin/man-cgi?00+00

[27] T. Blaschke, "Automated model generation for the lifecycle of android applications and the application of that model to an ifc-analysis," April 2014.

[28] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, Dec. 2009.

[29] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '95*, 1995.

ÍNDICE DE TABLAS

ÍNDICE DE GRÁFICAS