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

BOGOTA 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

BOGOTA 2015

Índice general

1. Introducción **6**

1.1. Contexto 6

2. Descripción del problema **10**

2.1. Trabajos Relacionados 12

2.1.1. JIF 12

2.1.2. JOANA 12

2.1.3. FlowDroid 13

2.1.4. TaintDroid, Dinamic Taint Tracking, para la detección de fugas de Información 14

2.1.5. Comparación de técnicas 15

2.1.6. Clasificación de Sources y Sinks 16

2.2. Propuesta 16

3. Diseño e Implementación **19**

3.1. Limitaciones técnicas para implementar el prototipo 19

3.2. Diseño de la solución 21

3.2.1. Definición de la política de seguridad 21

3.2.2. Consideraciones para verificar el cumplimiento de la política mediante Jif 21

3.2.3. Pasos para el diseño de la solución 22

3.2.4. Descripción implementación prototipo 24

4. Evaluación **25**

4.1. Consideraciones de evaluación 25

4.2. Evaluación conjunto de aplicaciones 25

5. Trabajo Futuro y Conclusiones **31**

5.1. Discusión 31

5.2. Trabajo Futuro 31

5.3. Conclusiones 31

6. Anexos **32**

6.1. Instrucciones para probar del prototipo 32

6.2. Instrucciones para uso de FlowDroid 33

ÍNDICE DE TABLAS

4.1. Descripción aplicaciones de prueba 26

4.2. Descripción aplicaciones de prueba 27

4.3. Descripción aplicaciones de prueba 28

4.4. Comparación Precisión entre FlowDroid y Prototipo. Los campos tF y tP, describen el tiempo que tarda Flowdroid y el Prototipo(respectivamente) en realizar el análisis para casos de prueba, evaluados. 29

ÍNDICE DE GRÁFICAS

2.1. Static Analisys Tool Diagrama interno. Ilustra la composición interna de la herramienta propuesta para el análisis estático de aplicaciones Android. 18

2.2. Static Analisys Tool. Ilustra el input esperado por la herramienta, y el resultado devuelto. 18

3.1. Pasos para el diseño de la solución. 22

3.2. Diseño de la solución paso 1. Ilustra las clases especificas de la clase de Android, anotadas manualmente. 23

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 dataflow 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 qué 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, 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 como 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. Contexto

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, 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 que puede alcanzar 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, es decir, 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 para las cuales 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, así, 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, a analizar durante la ejecución del programa.
Por otra parte, dependiendo de la finalidad con qué 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. Por el contrario, si se busca garantizar que a tiempo de ejecución la aplicación no incurrirá 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.
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.

En análisis estático, generalmente, se aplican técnicas de seguridad de tipado(Typed-Inference/Security-Typed Analysis) y técnicas de flujo de datos(Data/Control Flow Analysis). Con técnicas Security-Typed las propiedades de confidencialidad e integridad son anotadas en el código, y verificadas a tiempo de compilación para garantizar que se cumplen en tiempo de ejecución. Con las técnicas de flujo de control o flujo de datos, se hace seguimiento al flujo de los datos o control de flujo para verificar el cumplimiento de políticas de seguridad, generalmente se utilizan 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).

Al consultar la literatura científica, es posible inferir que parte importante de las propuestas para análisis de fuga de información en aplicativos Android(TaintDroid[2], Flow-Droid[3], DidFail[4], DroidForce[5]), parten del bytecode para realizar data-

flow analysis, mediante técnicas de análisis tainting. Tainting es un tipo especial de análisis de flujo de datos, que hace seguimiento al flujo de datos entre un conjunto de fuentes considerados privados y/o sensibles; y un conjunto de destinos considerados no confiables, sources y sinks, respectivamente.
Tales propuestas se enfocan en analizar aplicativos de terceros para detectar flujos de datos indebidos, y no: para garantizar el cumplimiento de determinadas políticas de seguridad. En consecuencia, es complejo que el desarrollador garantice la ausencia de fugas de información en la aplicación que implementa, partiendo de tales herramientas. Puesto que, al seguir únicamente a los datos marcados, los datos no marcados para el análisis, pueden acarrear fugas de información(under-tainting). Adicionalmente, si no se hace seguimiento al flujo de control pueden existir fugas de información a través de flujos implícitos, ya que, el análisis estará centrado en flujos explícitos.
No obstante, las limitaciones propias de un análisis basado en flujo de datos, pueden superarse enfocando el análisis de la aplicación hacia técnicas de análisis basadas en control de flujo de información, ya que estás analizan el aplicativo de forma estática para identificar todos los posibles caminos que podría tomar la aplicación en tiempo de ejecución. Así, con análisis basado en control de flujo de información, no sólo es posible prevenir fugas por under-tainting y flujos implícitos; sino que también, es posible ofrecer garantías del cumplimiento de determinadas políticas de seguridad.
Ahora, 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, una herramienta basada en lenguajes tipados de seguridad para realizar control de flujo de información, el inconveniente es que está implementada para aplicaciones en Java, y no para aplicativos Android.
En general, herramientas como estas, basadas en técnicas de análisis Security-Typed, implican conceptos como flujo de información, políticas de confidencialidad e integridad, y chequeo de tipos.
El flujo de información describe el comportamiento de un programa, desde la entrada de los datos hasta la salida de los mismos.
Confidencialidad e integridad son políticas de seguridad, aplicables mediante control de flujo de información. La confidencialidad busca prevenir que la información fluya hacia destinos no apropiados, mientras que, la integridad busca prevenir que la información provenga de fuentes no apropiadas. 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. Mientras que, con políticas de integridad, la finalidad es evitar modificación de la información, de forma no consentida.
Por consiguiente, verificar que un programa utilice la información acorde a tales políticas, implica analizar sus flujos de información, de inicio a fin. Para el 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.
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), regulador de 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[2], Flow-Droid[3], DidFail[4], DroidForce[5]), indica que la mayoría de propuestas hacen data-flow analysis mediante técnicas de análisis usando 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. Guiar el análisis de aplicaciones propias con el fin de verificar políticas de confidencialidad e integridad, bajo tales propuestas, puede implicar: mayor dificultad en el código a analizar, incompletitud en el análisis(under-tainting) y no detección de flujos implícitos. Esto debido a que, aún cuando el desarrollador conoce la funcionalidad de su propio código, las optimizaciones realizadas por el compilador pueden adicionar complejidad al mismo[6, pag. 43]; el seguimiento de los datos a través del programa está centrado en datos marcados, datos no marcados quedan fuera del análisis; flujos de datos a través de estructuras de control, por ejemplo, las sentencias if, permiten inferir valores de datos marcados como source, sin necesidad de generar flujos explícitos entre sources y sinks, los cuales si pueden ser detectados por las técnicas de análisis tainting.
Otra razón fundamental para no analizar aplicaciones propias con tales propuestas es que están diseñadas para detectar flujos de datos indebidos, y no para garantizar el cumplimiento de políticas de seguridad en una aplicación.
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 estás anotaciones durante el chequeo de tipos, verificando el cumplimiento de la propiedad de seguridad non-interference.

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[7]. 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[8], 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. 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[2.1.6], un clasificador automático de sources y sinks para la Api de Android.
FlowDroid provee un alto recall y precisión[3] 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[9], un framework que genera código intermedio para código Java y código ejecutable Android(dex). Adicionalmente, con el framework Heros[10], 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[11], 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[3, 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.

2.1.4. 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 marcas de archivo son utilizadas para verificar la persistencia de los datos, acorde a las políticas de seguridad.
Otra medida para reducir sobrecarga en la ejecución del análisis, consiste en no hacer seguimiento a flujos de control, generando no detección de flujos implícitos[2, 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.

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 flujos de datos.

2.1.5. 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 hacer seguimiento al flujo de datos a través del programa, o en verificar flujos 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 aplicar análisis mediante control de flujo de información, generalmente permiten definir anotaciones de seguridad en el código fuente de la aplicación, para verificar sus flujos de información. Estas generalmente se basan en técnicas de seguridad de tipado(Security-Typed Analyses), o en grafos que describen el comportamiento del programa, como Contol Dependence Graphs(PDG) y System Dependence Graphs(SDG). Ambas técnicas recurren a etapas de análisis de compilación, sin embargo, mientras las técnicas de Security-Typed sólo requieren llegar hasta el chequeo de tipos; las basadas en grafos de dependencia deben llegar hasta la representación de código intermedio para generar los respectivos grafos. Si bien, con grafos de dependencia se tiene mayor precisión en el análisis, su ejecución es costosa, ya que genera una complejidad de orden polinomial, O(N)3[12, page 3]. Las motivaciones para guiar el análisis bajo una u otra perspectiva, 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.

2.1.6. 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. Propuesta

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 confidencialidad.
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 confidencialidad.
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. Si bien, ambas analizan flujos de información en aplicaciones Java, y podrían ser extendidas para anotar código Android, las técnicas utilizadas por cada una 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. Mientras JOANA se enfoca en precisión, JIF posee un modelo de anotaciones (DLM) encargado de definir 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 a JIF un clasificador para sources y sinks de Android. El setup de JIF para Android consiste en implementar las adaptaciones necesarias para que el lenguaje JIF reconozca código de la API de Android, y admita anotaciones JIF dentro de código Android, pues aunque en esencia el código Android es código Java, JIF no tiene como saberlo. También se requiere la integración de un clasificador de sources y sinks al sistema de anotaciones de JIF, con el fin de proveer información necesaria para evaluar las políticas de confidencialidad.
La figura 2.1 expone los elementos necesarios para construir la herramienta de análisis. Básicamente, se requiere un módulo que extienda las clases en JIF para que el lenguaje reconozca código de la API de Android, es decir, para que admita anotaciones dentro del código Android: Setup extended JIF classes. Un módulo que integre el clasificador de sources y sinks de Android al sistema de anotaciones en JIF: Android Sources and Sinks. Adicionalmente, se requiere un modulo que evalúe las políticas de confidencialidad, Checking Rule Sets, que debe tener comunicación con los módulos anteriormente descritos. Como entrada, la herramienta recibe el código fuente de la aplicación, debidamente anotado por el desarrollador, y parte de las anotaciones definidas para retornar los resultados del análisis.
Habiendo realizado las extensiones necesarias, se espera contar con una herramienta de análisis de flujo de información, para un conjunto definido de clases en Android. En la figura 2.2 se ilustra el comportamiento esperado.

Luego, la estrategia de evaluación, consiste en verificar si la herramienta implementada identifica pérdida de información mediante detección de flujos implícitos. Esto debido a que, como se menciona en la descripción del problema, parte importante de las propuestas para detección de fuga de información en aplicaciones Android, hacen data-flow analysis aplicando técnicas de análisis tainting, y en contraste con las técnicas de análisis de flujo de información, las técnicas de análisis tainting no necesariamente consideran flujos implícitos. Por tanto, al estar basada en JIF, cuyo

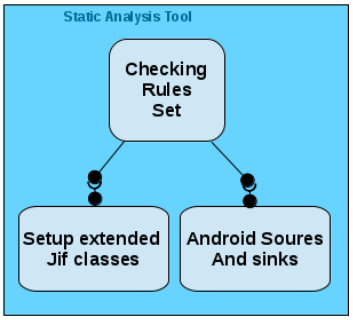

Figura 2.1: Static Analisys Tool Diagrama interno. Ilustra la composición interna de la herramienta propuesta para el análisis estático de aplicaciones Android.

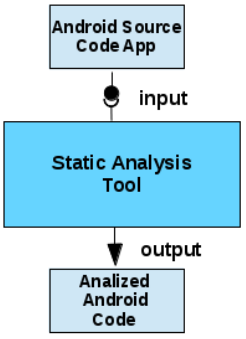

Figura 2.2: Static Analisys Tool. Ilustra el input esperado por la herramienta, y el resultado devuelto.

enfoque de análisis es precisamente flujo de información, se esperaría que la herramienta planteada esté en capacidad de reconocer flujos implícitos.

Más específicamente, se puede partir de DroidBench[11], el benchmark de Flowdroid[2.1.3], tomar el conjunto de aplicaciones con que prueban la detección de flujos implícitos, y analizarlas con la herramienta propuesta.
Finalmente estos resultados serían comparados con los obtenidos mediante otras herramientas para análisis de fuga de información en aplicaciones Android.

En este orden de ideas, la evaluación de la herramienta propuesta está enfocada en: medir recall frente a la detección de flujos implícitos, es decir, medir que no genere falsos negativos ante la existencia de fugas de información, provenientes de flujos implícitos.

Por último, cabe anotar que aunque la presente propuesta está centrada en verificar políticas de confidencialidad, en caso de contar con el tiempo prudente, sería interesante analizar políticas adicionales como por ejemplo, integridad y declasificación, pues estas son verificables mediante el modelo de evaluación de JIF, modelo del que parte la herramienta de evaluación planteada.

CAPÍTULO 3

Diseño e Implementación

3.1. Limitaciones técnicas para implementar el prototipo

-*Características del lenguaje Java no soportadas por jif:*
si bien, jif posibilita la evaluación de políticas de confidencialidad e integridad para aplicativos implementados en Java, mediante anotaciones que extienden el lenguaje, el manual de referencia de jif es claro en precisar características del lenguaje java no soportadas, 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, son lo primero que se ejecuta una vez se carga la clase, o si se trata de instance initialization blocks, se ejecutan cada vez que se crea una instancia de la clase) y threads.
Partiendo de estas precisiones, aplicaciones Android que presenten tales características son excluidas del grupo de aplicaciones a analizar mediante la herramienta propuesta.
Adicional a las limitaciones propias de jif frente al lenguaje Java, tras experimentar la anotación manual de una serie de aplicaciones Android, se identifican varias limitaciones técnicas para la anotación de código perteneciente a la API de Android. Entre las limitaciones identificadas están:
- *Sintaxis for-each:*
- *Símbolo para sobreescritura de métodos:*
jif no reconoce el símbolo de anotación @, utilizado para indicar que un método es sobreescrito(@Override). Según el componente que se esté implementando, la construcción de aplicaciones Android requiere la sobreescritura de métodos, así cuando se define una actividad, métodos del ciclo de vida como onCreate deben ser sobreescritos. La dificultad que se presenta está en que jif no reconoce el símbolo de anotación, y no en que jif no soporte la anotación de métodos, puesto que las pruebas realizas señalan que jif soporta tal característica. Ante esto, la solución que se adopta para la implementación del prototipo es comentar las líneas del programa que contengan @Override.
- *Casting entre tipos EditText y View:*
el framework de Android cuenta con diferentes clases para manejar las interfaces graficas 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. EditText permite adicionar campos de texto editables en interfaces UI. El casting entre los tipos de datos que representan ambas clases, suele usarse 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 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. Una forma de superar esta limitación es asingar el valor contenido en tipos EditText, a variables tipo String, de modo que no se requiera el casting descrito.
- *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 se soluciona definiendo una clase R que contenga los recursos definidos como variables y no como clases.
- *Sources y Sinks:*
en el diseño inicial de la solución, se planteó utilizar SuSi para clasificar los sources y sinks en las aplicaciones a analizar, sin embargo, partir del extenso conjunto de sources y sinks que SuSi clasifica para la API de Android, implica una mayor complejidad en el análisis, puesto que, en un aplicativo todo el código que le conforma puede hacer parte de sources o de sinks. Adicional a lo complejo que se puede tornar el análisis, los sources y sinks a considerar dependen de la política de seguridad a evaluar, en ese orden de ideas, partiendo de la política de seguridad que se defina, del listado proveído por SuSi, se selecciona un subconjunto específico de sources y sinks.

Además de las limitaciones descritas anteriormente, para las cuales se propuso una solución, se identifican otras en que la solución sobrepasa los límites de la presente investigación, puesto que corresponden a sintaxis no aceptada por el compilador de jif.

- *Sintaxis para hashmaps:*
- *Sintaxis para listas:*
- *Paso de statements dentro de los argumentos de un método ({}):*
- *No soporte para LinkedLIst:*

3.2. Diseño de la solución

3.2.1. 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.

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

Versión de la API de Android:
los experimentos previos a la implementación del prototipo y, la implementación del mismo se realiza partiendo de la versión Android 4.2.2(API Level 17).

Versión del compilador de jif:
se parte de la versión 3.4.2 del compilador de jif, para llevar a cabo tanto los experimentos previos como el análisis de las aplicaciones anotadas por el prototipo.

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 método como punto de entrada para iniciar su ejecución(el método main de la clase principal); una aplicación Android puede tener más de un punto de entrada, generados a partir de los diferentes tipos de componentes que la pueden integrar(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 ello cuenta con 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.

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 campo EditText, si y sólo si, es de tipo textPassword, es decir, un campo que almacena contraseñas.

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 poder ser conocida 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.

Evaluación del flujo de información:
Para evaluar el flujo de información, se tienen en cuenta todos los métodos definidos dentro de la clase, se asume que todos los métodos son invocados, esto con el fin de evitar omisiones de flujo de información a través de los canales previamente mencionados.

Estructura de trabajo en JIF:

3.2.3. Pasos para el diseño de la solución

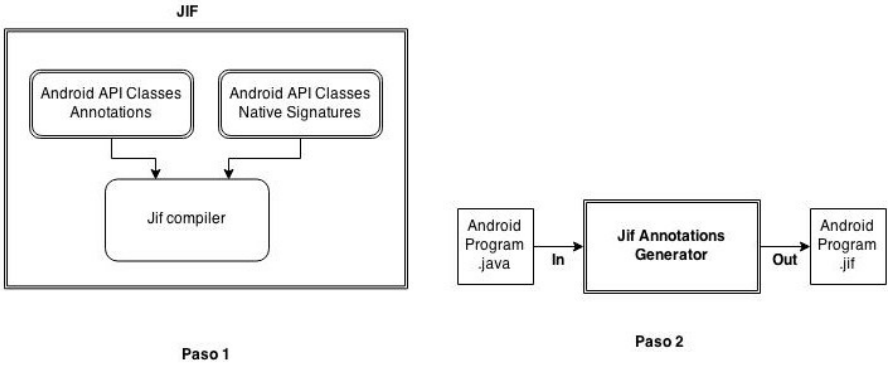

Figura 3.1: Pasos para el diseño de la solución.

-Paso uno: hacer que jif reconozca determinadas clases de la API de Android.
Adicional al mecanismo de anotación que se maneja en jif, en que para hacer análisis al flujo de información de un programa java, se debe implementar la versión del respectivo programa para jif. También es posible adicionar clases Java ya existentes, utilizando signaturas nativas para que el compilador jif las reconozca, esto es: generando una versión jif de la clase java, donde se declaran constructores y cuerpo de los métodos a utilizar de la clase fuente java.
Para el presente trabajo se utilizan ambas opciones de anotación. El criterio para decidir que se anota de una u otra forma, depende de lo que represente la clase Android para verificar la política de seguridad establecida. Las clases Log y SmsManager, que representan canales para conocer información, son anotadas de forma no nativa. La opción de anotación nativa se utiliza para librerías Android, por ejemplo

la clase TelephonyManager necesaria para utilizar el método getDeviceId.
A continuación se ilustran las clases anotadas.

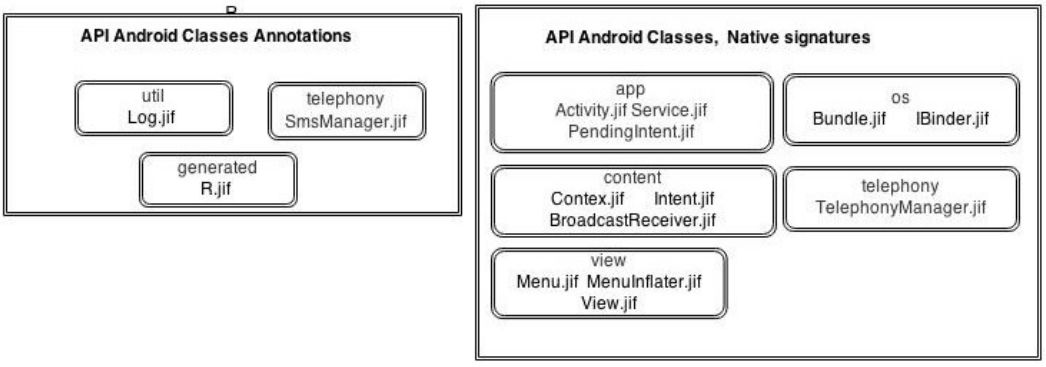

Figura 3.2: Diseño de la solución paso 1. Ilustra las clases especificas de la clase de Android, anotadas manualmente.

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.

-Paso dos: Definir Autoridades y forma de anotación del programa Android a analizar. Una clase Android tendrá una Autoridad máxima(un principal), en este caso Alice, así que, información con nivel de seguridad alto deberá pertenecer a dicha autoridad.
Jif hace seguimiento al flujo de información del programa, asociando un label de seguridad al program counter de cada sentencia y expresión del programa, program counter(pc) label. Este pc se ve afectado por el label de seguridad que se especifique en la declaración de variables y métodos.
La sintaxis para anotación de variables es:
type$\{L\}$ *varName;*
donde type especifica el tipo de dato que almacena la variable, {L} el label de seguridad para especificar quien es el dueño de la variable, y name, el respectivo nombre de la variable.
Definición de arrays:
en jif un array cuenta con 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.
Un método se escribe de la forma:
type $\{RTL\}$ *methodName* $\{BL\}$ *(arg1*$\{AL\}$*,,, argn*$\{AL\}$*)* :$\{EL\}$
RTL, Return Type Label, indica el label de seguridad con que queda el tipo de dato devuelto por el método.
BL begin label, representa el máximo nivel se seguridad del pc label 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 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 los AL con que han definido el método.

EL end label, indica el pc label en el punto de terminación del método, y representa la información que puede ser conocida.
Cuando un label no es especificado, Jif define unos por defecto. En el caso de RTL, jif hace un join entre los diferentes AL con que ha sido definido el método.

Partiendo de que Jif se fundamenta en labels de seguridad para hacer seguimiento al flujo de información del programa, es necesario definir los labels a anotar para métodos y variables del programa.
En el caso de variables con nivel de seguridad alto, la anotación debe ser:
type{Alice:} varName;
Para el resto de variables, entran a jugar las anotaciones definidas por Jif acorde al contexto donde están definidas.

Ahora en el caso de los métodos, la anotación varía acorde a si el método debe influenciar(acceder, modificar) o no, información anotada con nivel de seguridad alto.
Partiendo de lo anterior, se define un algoritmo de anotación que se condensa en un generador de anotaciones.
- Descripción criterios de anotación:
Definición A: anotación de variables con nivel de seguridad alto:
modifier type{Alice:} varName;
Definición B: métodos que se sobreescriben. El sistema de anotaciones de jif exige que el nivel de seguridad del método desde donde se invoca la sobreescritura de un método, no debe ser menos restrictivo que el método a sobreescribir, y como se mencionó al final del paso uno, los métodos a sobreescribir deben poder ser invocados desde todo programa Android, siguiendo con este principio, y buscando que jif no limite el flujo de información, estos métodos deben ser anotados con BL público({}).
Definición C: anotación de métodos con sources
Los labels para la definición del método(BL, EL, AL)se anotan de la siguiente manera:
modifier type nameMethod{*Alice:*} type{*Alice:*} (arg1,.....type{*Alice:*}argn) {}
Si dentro del método se definen arrays, sus respectivos BL y SL, deben ser anotados así: modifier type{Alice:}[ ]{Alice:}
Definición D: anotación de métodos que no reciben información del source. nameMethod{}(type{*Alice:*}arg1,.....type{*Alice:*}argn) {}
- Pasos:
(1) Identificar sources de la clase. Si se encuentran sources continuar con los pasos (2) a (4), sino, continuar con paso (2) y aplicar definiciones B y D.
(2) Identificar el total de métodos de la clase.
(3) Del total de métodos listar los que son invocados con el source.
(4) Del total de métodos listar los que no son invocados con el source.
(5) Aplicar definición C a listado del paso(3).
(6) Aplicar definición D a listado del paso (4).
(7) Aplicar definición B.
(8) Aplicar definición A a listado del paso (1).

3.2.4. Descripción implementación prototipo

clases, qué aporta cada clase.

CAPÍTULO 4

Evaluación

Ventajas y limitaciones de la solución.
Si aplica, evaluación de desempeño.
Si aplica, evaluación de usabilidad. Hay otras soluciones similares?
Cuáles son las diferencias y las ventajas y desventajas con respecto a esas soluciones.

4.1. Consideraciones de evaluación

No se consideran flujos de información vía interAppComunicación. Por ejemplo, varias aplicaciones que se comunican entre sí.

4.2. Evaluación conjunto de aplicaciones

Para la evaluación del prototipo se toma un grupo de testcases de DroidBech, el benchmark de FlowDroid, aplicables para la evaluación de la política de seguridad establecida.
Se considera con nivel de seguridad alto, variables y métodos que almacenan y modifican(respectivamente), información considerada como privada(Sources).
Se considera con nivel de seguridad bajo, canales para envío de mensajes, muestra de logs y canales creados durante el flujo del programa.
A continuación se describen los 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 y la técnica de análisis propuesta en el presente trabajo.

Cuadro 4.1: 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

Cuadro 4.2: 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

Cuadro 4.3: 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

La tabla4.4 presenta los resultados de analizar los casos de prueba previamente descritos, tanto con FlowDroid como con el prototipo. Los resultados se califican como: Falso Positivo(FP) cuando se detecta un leak que no existe; Falso Negativo(FN) cuando no se detecta un leak existente; Verdadero Positivo(TP) cuando se detecta un leak existente; Verdadero Negativo(TN) cuando no existe leak que detectar.
El tiempo que tarda cada herramienta para evaluar el respectivo testcase es medido con el comando *time*.

Cuadro 4.4: Comparación Precisión entre FlowDroid y Prototipo. Los campos tF y tP, describen el tiempo que tarda Flowdroid y el Prototipo(respectivamente) en realizar el análisis para casos de prueba, evaluados.

Testcase	**Leaks**	**FlowDroid**	**Prototipo**	**t F**	**t P**
AndroidSpecific_DirectLeak1	1	TP	TP	5.371s	2.063s
AndroidSpecific_InactiveActivity	0	TN	FP	3.255s	2.469s
AndroidSpecific_LogNoLeak	0	TN	TN	5.505s	2.946s
AndroidSpecific_Obfuscation1	1	TP	TP	6.734s	2.706s
AndroidSpecific_PrivateDataLeak2	1	TP	TP	6.144s	2.644s
ArraysAndLists_ArrayAccess1	0	FP	FP	4.708s	1.278s
ArraysAndLists_ArrayAccess2	0	FP	FP	4.4s	1.361s
GeneralJava_Exceptions1	1	TP	TP	6.397s	2.755s
GeneralJava_Exceptions2	1	TP	TP	5.887s	1.980s
GeneralJava_Exceptions3	0	FP	FP	6.008s	2.032s
GeneralJava_Exceptions4	1	TP	TP	5.731s	2.313s
GeneralJava_Loop1	1	TP	TP	5.605s	2.800s
GeneralJava_Loop2	1	TP	TP	4.719s	1.361s
GeneralJava_UnreachableCode	0	TP	FP	3.792s	1.197s
ImplicitFlows_ImplicitFlow1	1	FN	TP	4.853s	1.331s
ImplicitFlows_ImplicitFlow2	1	FN	TP	4.496s	1.212s
ImplicitFlows_ImplicitFlow4	1	FN	TP	4.375s	1.224s
Lifecycle_ActivityLifecycle3	1	TP	TP	4.792s	1.222s
Lifecycle_BroadcastReceiverLifecycle1	1	TP	TP	4.456s	1.061s
Lifecycle_ServiceLifecycle1	1	TP	TP	5.225s	1.180s

-*Resultados de precisión*
En lo que respecta a los resultados del Prototipo, los FP correspondientes a AndroidSpecific_InactiveActivity y GeneralJava_UnreachableCode, surgen como consecuencia de realizar el análisis asumiendo que el desarrollador utiliza lo que implementa.
Por otro lado, en el caso de ArraysAndLists_ArrayAccess1 y ArraysAndLists_ArrayAccess2, 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.
La detección de flujos implícitos podría ser un elemento diferenciador.
-*Resultados de desempeño*
Se podría destacar como positivo que el análisis de flujo de información mediante técnicas de tipado de seguridad, requiere menos tiempo que la técnica de marcado

de datos utilizada por FlowDroid.

-*Acerca de por qué FlowDroid no detecta flujos implícitos*
El análisis de FlowDroid utiliza técnicas DataFlow, específicamente, utiliza tainting análisis. Para hacer seguimiento al flujo de información de un programa, la técnica de análisis tainting se basa en: asociar una o más marcas con el valor de los datos en el programa, y en propagarlas. Dependiendo de los criterios definidos, la marca puede ser propagada a causa de flujos explícitos o de flujos implícitos. En flujos explícitos la propagación ocurre cuando el valor de una variable marcada está implicada en el calculo de otra varibale. 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 varible.
En el caso de FlowDroid el análisis se concentra en el marcado de flujo de datos, esto significa que la herramienta está en capacidad de detectar flujos explicitos, pero no flujos implicítos generados a través del flujo de control del programa.

-*Qué tanto cambia la anotación del código original* - adición de catch.

CAPÍTULO 5

Trabajo Futuro y Conclusiones

5.1. Discusión

Límites de la solución propuesta

5.2. Trabajo Futuro

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

5.3. Conclusiones

Qué aprendimos con este trabajo.

CAPÍTULO 6

Anexos

6.1. Instrucciones para probar del 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 devuleve 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.

6.2. 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] 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.

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

[4] 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.

[5] 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.

[6] B. Chess and J. West, *Secure Programming with Static Analysis*. Gary McGraw, June 2007.

[7] 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), Springer Berlin / Heidelberg*, pp. 123–138, February 2013.

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

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

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

[11] EC SPRIDE. (2014, June) Droidbench – benchmarks. [Online]. Available: http://sseblog.ec-spride.de/tools/droidbench/

[12] 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.