

ANÁLISIS DE FLUJOS DE INFORMACIÓN EN APLICACIONES ANDROID

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

Resumen—

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

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

CATEGORÍAS Y DESCRIPCIÓN DE TEMÁTICAS

Análisis de flujos de información en aplicaciones.

TERMINOS GENERALES

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

PALABRAS CLAVE

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

I. INTRODUCCIÓN

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

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

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

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

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

El Artículo está organizado de la siguiente manera: XXX

II. CONTEXTO

II-A. Técnicas de análisis de código

II-A1. Análisis estático y dinámico: Las soluciones propuestas para detectar fuga de información en aplicaciones Android, se enmarcan en el análisis estático o dinámico de la aplicación, en algunos casos, se combinan ambos tipos.

En **análisis estático**[2], se estudia el código del programa para inferir todos los posibles caminos de ejecución. Esto se logra construyendo modelos de estado del programa, y determinando

los estados posibles a alcanzar por el programa. No obstante, debido a que existen múltiples posibilidades de ejecución, se opta por construir un modelo abstracto de los estados del programa. La consecuencia de tener un modelo aproximado es pérdida de información y posibilidad de menor precisión en el análisis.

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

II-A2. Técnicas utilizadas en análisis estático: Generalmente, para verificar el cumplimiento de políticas de seguridad mediante análisis estático, se aplican técnicas de seguridad de tipado (Typed-Inference/Security-Typed Analysis) y técnicas de flujo de datos(Data/Control Flow Analysis)[3].

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

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

II-A3. Security Typed Languages: Las herramientas basadas en técnicas de análisis Security-Typed, involucran conceptos como flujo de información, políticas de confidencialidad e integridad, y chequeo de tipos. *Flujo de información:* el flujo de información describe el comportamiento de un programa, desde la entrada de los datos hasta la salida de los mismos.

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

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

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

El compilador realiza el chequeo de tipos, partiendo del

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

II-B. Background

II-B1. 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)[5], verificando que los flujos de información generados cumplen con las restricciones establecidas.

II-B2. DML(Decentralized Label Model): Jif basa su sistema de anotaciones en el modelo de etiquetas DLM, 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, Chuck, 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.

II-B3. 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, program-counter label(pc). En cada punto del programa, el (pc) representa la información que podría conocerse tras la ejecución de ese punto del programa. El (pc) es afectado por 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.

III. DESCRIPCIÓN DEL PROBLEMA

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

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

Las motivaciones para este tipo de acciones varían acorde al tipo de información, por ejemplo: monitorear información de ubicación para mostrar publicidad no solicitada; seguir las acciones sobre el dispositivo, para conocer qué aplicaciones son rentables de desarrollar, o para ayudar a aplicaciones maliciosas a evadir defensas; acceder a información de cuentas del usuario con fines delictivos; obtener información de contactos y calendario del usuario, buscando modificar los datos; obtener información del celular (número, estado, registro de MMS y SMS) para interceptar llamadas y enviar mensajes sin consentimiento del usuario.

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

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

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

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

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

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

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

Por el otro, aún cuando se hace análisis tainting de forma estática, y el marcado de datos puede ser propagado para todos los caminos posibles de ejecución del programa, superando el inconveniente de under-tainting, la detección de flujos implícitos es posible si, en la construcción de la herramienta de análisis se propaga el marcado de datos para flujos implícitos[12]. Sin embargo, las propuestas que basan su análisis en data-flow estático, suelen restringir la propagación del marcado de datos a flujos explícitos, ganando eficiencia en el análisis. Las propuestas mencionadas anteriormente, no son ajenas a tal generalidad (DidFail[8][page 33], FlowDroid[7][page 30]).

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

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

IV. TRABAJOS RELACIONADOS

Dentro de las propuestas existentes están

V. PROPUESTA

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

La propuesta para cumplir con tales requerimientos consiste en proveer una herramienta de análisis de flujo de información, mediante el sistema de anotaciones de Jif. Se parte de Jif porque ofrece un sistema de anotaciones basado en un modelo de etiquetas(DLM), más un módulo de verificación(compilador de Jif), tal como se ilustra en la figura 1. Así, mediante las etiquetas del DLM el desarrollador puede definir políticas de seguridad en el código de su aplicación, para luego validar con el módulo de verificación, si la aplicación respeta tales políticas.

Ahora bien, el diseño ideal para contribuir con la solución del problema es: una herramienta que contenga el setup de Jif para Android, e integre un clasificador de sources y sinks. De modo que, la herramienta analice flujos de información en aplicativos Android, verificando que cumplan con las políticas de seguridad que el desarrollador ha definido.

El Setup de Jif para Android consiste en las clases de la API Android con anotaciones Jif(Anotaciones a la API), estas anotaciones son necesarias porque la API permite que la aplicación escriba o lea información de los sources y los sinks, generando flujos de información entre estos. En consecuencia, para analizar el flujo de información entre sources y sinks se requiere anotar las clases de la API.

No obstante, para efectos del presente trabajo se limita el Setup de Jif, partiendo de una política de seguridad específica. Por consiguiente, el diseño se centra en soportar un conjunto reducido de clases de la API de Android, y en incluir un conjunto específico de sources y sinks; de acuerdo a una política de seguridad establecida. Ese conjunto de sources y sinks, se toma del listado de sources y sinks proveído por SuSi ??.

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

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

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

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

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

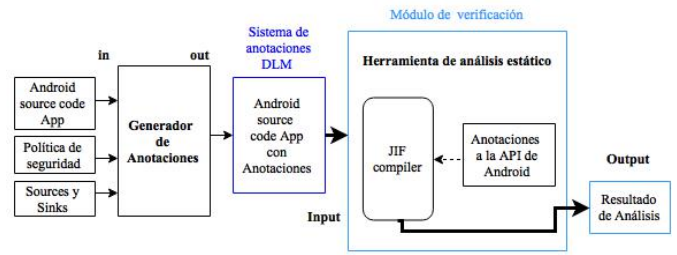


Figura 1: Diseño herramienta de análisis estático.

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

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

V-A. Definición de la política de seguridad

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

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

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

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

V-B. Lineamientos de anotación

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

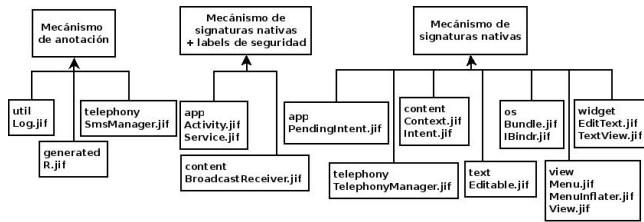


Figura 2: Mecanismos de anotación para clases de la API.

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

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.

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

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

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

V-B2. Anotaciones a la API de Android: Para controlar el flujo de información que se envía hacia sinks, 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 desde el punto del programa donde es invocado el método, y el nivel de seguridad de los argumentos con que se invocan tales métodos.

Para ello se recurre a los principios que regulan el llamado a métodos en el sistema de anotaciones de Jif, entre los cuales está que, el nivel de seguridad de los argumentos con que se invoca un método, debe ser menor o igual de restrictivo que los labels de seguridad con que se ha definido el método. Tomando como ejemplo el método `sendMessage` de la clase `SmsManager`, mediante el cual se envían mensajes de texto:

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

Se tiene que el parámetro *text* es el que recibe la información a 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 un flujo de información indebido, dando lugar a un 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 `sendMessage` y aplicando lo anteriormente descrito, este método se anota de la siguiente manera:

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

VI. EVALUACIÓN

VII. TRABAJO FUTURO

VIII. CONCLUSIONES

REFERENCIAS

- [1] McAfee. (2014, February) Who's watching you?, mcafee mobile security report. [Online]. Available: <http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf>
- [2] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *WODA 2003 International Conference on Software Engineering (ICSE) Workshop on Dynamic Analysis*, ser. ICSE'03, Portland, Oregon, 2003, pp. 25–28. [Online]. Available: <http://www.cs.nmsu.edu/~jcCook/woda2003/>
- [3] S. Genaim and F. Spoto, "Information flow analysis for java bytecode," *Proceeding VMCAI'05 Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, 2005.
- [4] AndreiSabelfeld and A. wC.Myers, "Language-based information-flow security," *IEEE Journal*, vol. 21, no. 1, pp. 1–15, January 2003.
- [5] Cornell University. (2014, March) Jif reference manual. [Online]. Available: <http://www.cs.cornell.edu/jif/doc/jif-3.3.0/language.html#unsupported-java>
- [6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *9th USENIX Symposium on Operating Systems Design and Implementation(OSDI'10)*, pp. 1–15, October 2010.
- [7] C. Fritz, "Flowdroid: A precise and scalable data flow analysis for android," Master's thesis, Technische Universität Darmstadt, July 2013.
- [8] A. S. Bhosale, "Precise static analysis of taint flow for android application sets," Master's thesis, Heinz College Carnegie Mellon University Pittsburgh, PA 15213, May 2014.
- [9] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "Droidforce: Enforcing complex, data-centric, system-wide policies in android," *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*, pp. 1–10, September 2014.
- [10] C. Hammer and G. Snelting, "Formal characterization of illegal control flow in android system," *Signal-Image Technology Internet-Based Systems (SITIS), 2013 International Conference on*, pp. 293 – 300, Dec. 2013.

- [11] B. Yadegari and S. Debray, "Bit-level taint analysis," *2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014.
- [12] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *ISSTA '07 Proceedings of the 2007 international symposium on Software testing and analysis*, July 2007, pp. 196–206.