Non-Interference Preserving Compilation of Android Bytecode

Hendra Gunadi[1], Alwen Tiu[2], and Rajeev Goré[1]

[1] Research School of Computer Science, The Australian National University
[2] School of Computer Science and Engineering, Nanyang Technological University

Abstract. We aim to develop a proof-carrying code (PCC) framework to certify compliance of an Android application with a given information flow policy, specified via certain non-interference properties. In the PCC framework, a certified application comes with a statement of an information flow policy and a proof that its execution complies with the given policy. We follow a type-based approach for enforcing non-interference, in which typeable programs are guaranteed to be non-interferrent and typing derivations serve as certificates of non-interference. Our eventual goal is to produce a compiler tool chain that can help developers to develop Android applications that complies with a given policy, and automate the process of generating the final non-interference certificates for Dalvik bytecode. Android apps are typically developed in Java, and go through two stages of compilation. The first one compiles the Java code into JVM bytecode, and the second one compiles the JVM bytecode into the final Android Dalvik bytecode. Non-interference type systems exist for all three languages (Java, JVM and Dalvik), and Barth et. al. have shown that typeability is preserved going from Java source to JVM. To complete the picture, we show here that the compilation from JVM to Dalvik, as implemented in the official Android dx tool, preserves typeability. We then show how such a type-based certification scheme can be implemented, and give details of the certificate encodings in Android apps.

1 Introduction

Android is an operating system that has been used in many mobile devices. According to [2], Android has the largest market share for mobile devices, making it an attractive target for malware, so verification of the security properties of Android apps is crucial. To install an application, users can download applications from Google Play or third-party app stores in the form of an Android Application Package (APK). Each of these applications runs in an instance of a Dalvik virtual machine (VM) on top of the Linux operating system. Contained in each of these APKs is a DEX file containing specific instructions [1] to be executed by the Dalvik VM, so from here on we will refer to these bytecode instructions as DEX instructions. The Dalvik VM is a register-based VM, unlike the Java Virtual Machine (JVM) which is a stack-based VM. Dalvik is now superseded

by a new runtime framework called ART, but this does not affect our analysis since both Dalvik and ART use the same DEX instructions.

We aim at providing a framework for constructing trustworthy apps, where developers of apps can provide guarantees that the (sensitive) information the apps use is not leaked outside the device without the user's consent. The framework should also provide a mean for the end user to verify that apps constructed using the framework adhere to their advertised security policies. This is, of course, not a new concept, and it is essentially a rehash of the (foundational) proof carrying code (PCC) [24, 3], applied to the Android setting. We follow a type-based approach for restricting information flow [**?**] in Android apps. Semantically, information flow properties of apps are specified via a notion of non-interference [**?**]. In this setting, typeable programs are guaranteed to be non-interferrent, with respect to a given policy, and typing derivations serve as certificates of non-interference. Our eventual goal is to produce a compiler tool chain that can help developers to develop Android applications that complies with a given policy, and automate the process of generating the final non-interference certificates for DEX bytecode.

An Android application is typically written in Java and compiled to Java classes (JVM bytecode). Then using tools provided in the Android Software Development Kit (SDK), these Java classes are further compiled into an Android application in the form of an APK. One important tool in this compilation chain is the dx tool, which will aggregate the Java classes and produce a DEX file to be bundled together with other resource files in the APK. Non-interference type systems exist for Java source code [**?**], JVM [5] and (abstracted) DEX bytecode [22]. To build a framework that allows end-to-end certificate production, one needs to study certificate translation between these different type systems. The connection between Java and JVM type systems for non-interference has been studied in [4]. In this work, we fill the gap by showing that the connection between JVM and DEX type systems. Our contributions are the following:

- We give a formal account of the compilation process from JVM bytecode to DEX bytecode as implemented in the official dx tool in Android SDK. Section 3 details some of the translation processes.
- We provide a proof that the translation from JVM to DEX preserves typeability. That is, JVM programs typeable in the non-interference type system for JVM translates into typeable programs in the non-interference type system for DEX.
- We provide a proof-of-concept implementation of a certification framework for (non-optimized) Android applications. Our certificate for non-inteference (in the form of a typing derivation) can be injected into existing application structure for Android (the apk file) without requiring any changes to the Android framework or the Android app store. Our typechecker on the client side is a normal Android app that will run in most Android platforms.

The rest of the paper is organized as follows: In Section 2, we give an overview of JVM and DEX type systems for enforcing non-interference. Section 3 discusses

the compilation process from JVM bytecode to DEX bytecode. Section 4 discusses how to translate certificates from JVM to DEX. Section 5 shows a proof-of-concept implementation of our PCC framework and discusses some details of the stuctures of the apps and the certificates.

Due to space limit, we discuss only a limited fragment of JVM and DEX in this paper, but our results extend to a richer fragment of JVM, covering all fragments considered in [5]. The extended version of this paper containing detail technical results can be found in [18]. The prototype implementation discussed in Section 5 is available online.[3]

Related work. The cloest to our work is the Cassandra project [22, 23], that aims at developing certified app stores, where apps can be certified, using an information-flow type system similar to ours, for absence of specific information flow. Specifically, the authors of [22, 23] have developed an abstract Dalvik language (ADL), similar to Dalvik bytecode, and a type system for enforcing non-interference properties for ADL. Our type system for Dalvik is essentially that of Cassandra, modulo some minor differences in the language constructs. We choose to deal directly with Dalvik rather than ADL since we aim to eventually integrate our certificate compilation into existing compiler tool chains for Android apps, without having to modify those tool chains.

To deal with information flow properties in Android, there are several works addressing the problem [7, 17, 9, 12, 26, 20, 16, 19, 15, 13, 14] although some of them are geared towards the privilege escalation problem. Due to space limitation, we omit detail comparisons here, but the interested reader may consult the technical report [18] for further comments. We note only here that none of those work mentioned above deals with the problem of non-interference preserving compilation between JVM and DEX bytecode.

2 Information flow type systems for JVM and Android

In this section, we give an overview of Barthe et. al's type system for JVM [5] and a type system for Android DEX bytecode similar to that in [23]. Due to space constraints, and since our main purpose is to prove the non-interference-preserving compilation of JVM (which is a stack-based machine) to DEX (which is a register-based machine), we shall only give an overview of the main distinguishing features between the two abstract machines in this overview. In particular, we highlight only the potential information flow induced by the use of operand stack and how the type system of Barth et al prevents it, and how this would map to the type system for the register-based Dalvik machine. We therefore restrict to the simplest fragment of JVM which still exhibits some of the subtlety in proving non-interference, namely the fragment that covers basic arithmetic and control operators. We note however, our approach extends to richer fragments of JVM, including arrays, objects, method invocations and

[3] At `http://users.cecs.anu.edu.au/~hengunadi/TranslationProof.html`.

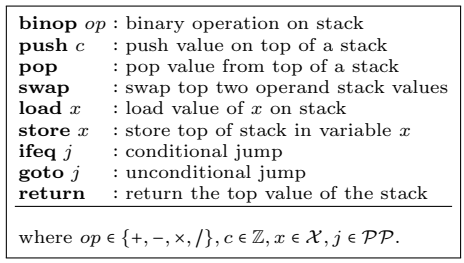

Fig. 1: JVM Instruction List

exceptions. The full technical details of these extensions can be found in the accompanying techical report [18].

2.1 A type system for JVM

A program P is given by its list of instructions. To simplify presentation, we consider here only instructions within a method, and do not consider explicitly method fields and objects. We assume the reader is familiar with elements of a stack frame in JVM, which consists of local variable slots, parameters and an operand stack (henceforth referred to simply as the stack). We consider here the following (idealized) instructions of JVM, given in Figure 1. We have abstracted away from various types of the same instructions (e.g., iload for loading integers, lload for loading long integers, etc) for simplicity. The actual implementation (see Section 5) does work directly on JVM instructions.

The set of local variables in a method is denoted by $\mathcal{X}$. The set of values, denoted by $\mathcal{V}$, is defined as $\mathcal{V} = \mathbb{Z} \bigcup \mathcal{L} \bigcup \{null\}$, where $\mathcal{L}$ is an (infinite) set of locations and $null$ denotes the null pointer. The set of program points is denoted by $\mathcal{PP}$. For any set X, we use the notation X^* to stand for a stack of elements from X. If st is stack, we write st_i to denote the i-th element of st.

The operational semantics of JVM, given in Figure 2, is formalized as a relation between a JVM state and another JVM state, or a value if the computation terminates. A JVM state is a tuple $\langle i, \rho, os \rangle$ where $i \in \mathcal{PP}$ is the program counter that points to the next instruction to be executed; $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$ is a partial function from local variables to values, and $os \in \mathcal{V}^*$ is an operand stack. Given a mapping ρ, we write $\rho \oplus \{x \mapsto v\}$ to denote another mapping such that $\rho \oplus \{x \mapsto v\}(x) = v$ and $\rho(y) = \rho \oplus \{x \mapsto v\}(y)$ for $y \neq x$. The operational semantics is parameterized by a program P. In the figure, we write $P[i]$ to denote the i-th instruction in program P, and $\underline{\text{op}}$ denotes the standard interpretation of the arithmetic operation op on integers.

The small-step operational semantics induces a successor relation between program points. Intuitively, one program point is a successor of another if they can be related by the small-step relation for some states. Given a program P, the successor relation induced by P, denoted by $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$, is defined as follows: for a program points i, its successor is defined based on $P[i]$:

$$\frac{P[i] = \textbf{push}}{\langle i, \rho, os\rangle \rightsquigarrow \langle i+1, \rho, n :: os\rangle} \qquad \frac{P[i] = \textbf{pop}}{\langle i, \rho, v :: os\rangle \rightsquigarrow \langle i+1, \rho, os\rangle} \qquad \frac{P[i] = \textbf{return}}{\langle i, \rho, v :: os\rangle \rightsquigarrow v, h}$$

$$\frac{P[i] = \textbf{goto}\ j}{\langle i, \rho, os\rangle \rightsquigarrow \langle j, \rho, os\rangle} \qquad \frac{P[i] = \textbf{ifeq}\ j \quad n \neq 0}{\langle i, \rho, n :: os\rangle \rightsquigarrow \langle i+1, \rho, os\rangle} \qquad \frac{P[i] = \textbf{ifeq}\ j \quad n = 0}{\langle i, \rho, n :: os\rangle \rightsquigarrow \langle j, \rho, os\rangle}$$

$$\frac{P[i] = \textbf{store}\ x \quad x \in \textbf{dom}(\rho)}{\langle i, \rho, v :: os\rangle \rightsquigarrow \langle i+1, \rho \oplus \{x \mapsto v\}, os\rangle} \qquad \frac{P[i] = \textbf{load}\ x}{\langle i, \rho, os\rangle \rightsquigarrow \langle i+1, \rho, \rho(x) :: os\rangle}$$

$$\frac{P[i] = \textbf{binop}\ op \quad n_2\ \underline{op}\ n_1 = n}{\langle i, \rho, n_1 :: n_2 :: os\rangle \rightsquigarrow \langle i+1, \rho, n :: os\rangle} \qquad \frac{P[i] = \textbf{swap}}{\langle i, \rho, v_1 :: v_2 :: os\rangle \rightsquigarrow \langle i+1, \rho, v_2 :: v_1 :: os\rangle}$$

Fig. 2: The operational semantics of (selected) JVM instructions

$$\frac{P[i] = \textbf{load}\ x}{i \vdash st \Rightarrow \left(\vec{k_a}(x) \sqcup se(i)\right) :: st} \qquad \frac{P[i] = \textbf{store}\ x \quad se(i) \sqcup k \leq \vec{k_a}(x)}{i \vdash k :: st \Rightarrow st}$$

$$\frac{P[i] = \textbf{swap}}{i \vdash k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st} \qquad \frac{P[i] = \textbf{ifeq}\ j \quad \forall j' \in \textbf{region}(i), k \leq se(j')}{i \vdash k :: st \Rightarrow \textbf{lift}_{\mathbf{k}}(st)}$$

$$\frac{P[i] = \textbf{goto}\ j}{i \vdash st \Rightarrow st} \qquad \frac{P[i] = \textbf{push}\ n}{i \vdash st \Rightarrow se(i) :: st}$$

$$\frac{P[i] = \textbf{binop}\ op}{i \vdash k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2 \sqcup se(i)) :: st} \qquad \frac{P[i] = \textbf{return} \quad se(i) \sqcup k \leq k_r}{i \vdash k :: st \Rightarrow}$$

Fig. 3: Selected JVM Transfer Rules

- If $P[i] = \textbf{goto}\ j$ then $i \mapsto j$.
- If $P[i] = \textbf{ifeq}\ j$, then $i \mapsto i+1$ and $i \mapsto j$.
- If $P[i] = \textbf{return}$ then i has no successor.
- If $P[i]$ is any instruction other than the above, then $i \mapsto i+1$

We write $i \mapsto$ when i has no successors.

The purpose of the type system is to make sure that typeable programs adhere to a given information flow policy, expressed in terms of security levels of input and output of a method or a program. Security levels are formalized as a lattice $(\mathcal{S}, \leq)$, where $\mathcal{S}$ is the carrier set and $\leq$ is the order relation of the lattice. Given $s, t \in \mathcal{S}$, we denote with $s \sqcup t$ the least upper bound of s and t. Throughout this paper, we shall assume implicitly that $\mathcal{S}$ is a two-element lattice, designated with H (high security level) and L (low security level) with the ordering $L \leq H$.

Program variables, fields and return values will be associated with certain security levels, and one is interested in showing that information does not flow from a source to a sink with a lower level security level. Formally the absence of such flow will be established via a notion of non-interference: one first fixes a security level of an observer, capable of observing variables or return values with the same or lower security levels, and shows that varying the values of variables or fields with higher security levels than the observer security level does not affect the observable values. Soundness of the type system is then expressed via the non-interference property, i.e., typeable programs are non-interferent.

In a stack-based machine such as JVM, the operand stack can hold intermediate values during computation, and hence to prevent indirect information flow, the security levels of elements of the operand stack needs to be tracked for each instruction in a program. The type system for JVM is thus specified as a judgment relating two stack types, one before the execution of an instruction, and one after. To check for implicit flow via conditional branching, two notions are introduced into the type system. One is the so-called *control dependence region* (CDR), that keeps track of points of programs under the influence of a guard, and the *security environment*. A security environment is a function $se : \mathcal{PP} \to \mathcal{S}$ which maps program points to security levels. A CDR is defined in terms of two functions: **region** and **jun**. The function $\mathbf{region} : \mathcal{PP} \to \wp(\mathcal{PP})$ can be seen as all the program points executing under the guard of the instruction at the specified program point, i.e. in the case of $\mathbf{region}(i)$ the guard will be program point i. The function $\mathbf{jun}(i)$ can be seen as the nearest program point which all instructions in $\mathbf{region}(i)$ have to execute (junction point). A CDR is safe if it satisfies the following SOAP (Safe Over APproximation) properties.

Definition 1. *A CDR structure (region, jun) satisfies the SOAP properties if the following properties hold :*

SOAP1. $\forall i, j, k \in \mathcal{PP}$ *such that* $i \mapsto j$ *and* $i \mapsto k$ *and* $j \neq k$ *(i is hence a branching point),* $k \in \mathbf{region}(i)$ *or* $k = \mathbf{jun}(i)$.

SOAP2. $\forall i, j, k \in \mathcal{PP}$ *, if* $j \in \mathbf{region}(i)$ *and* $j \mapsto k$*, then either* $k \in \mathbf{region}(i)$ *or* $k = \mathbf{jun}(i)$.

SOAP3. $\forall i, j \in \mathcal{PP}$ *, if* $j \in \mathbf{region}(i)$ *and* j *is a return point then* $\mathbf{jun}(i)$ *is undefined.*

We assume that every program comes with its security policy, expressing the security levels of the local variables and the return value of each of its methods (for simplicity we assume that every method returns a value). Formally, for each method in the program, its security policy comes in the form of a *method signature*: $\vec{k_a} \to k_r$ where $\vec{k_a} = \{x_1 : k_1, \ldots, x_n : k_n\}$, $k_i \in \mathcal{S}$ is the security level of the variable x_i, and k_r is the security level of the return value. Given such a method signature, we write $\vec{k_a}(x_i)$ to denote k_i. We assume that every program P comes with a table Γ of its methods' signatures. A *security stack* is a stack whose elements are security levels.

Formally, given a program P, the typing judgment takes the form:

$$\mathbf{region}, se, sgn, i \vdash st \Rightarrow st'$$

where i is the current program point; (**region**, **jun**) is a CDR; se is a security environment; sgn is method signature of the current method; and st and st' are security stacks. The typing judgment expresses the relation betwen security stacks, before and after the execution of the instruction $P[i]$. The typing rules are sometimes also called *transfer rules*. Figure 3 shows a selected set of transfer rules for JVM. In the figure, to simplify presentation, some parameters are left implicit, i.e., the method signature $\vec{k_a} \to k_r$, the CDR $(\mathbf{region}, \mathbf{jun})$, and the security environment se.

In the typing rule for **ifeq**, the stack type $\mathbf{lift_k}(st)$ denotes a stack type obtained from st by letting $\mathbf{lift_k}(st)_i = st_i \sqcup k$. Lifting of the stack types is needed to ensure no information leakage through stack operations such as pop or swap that do not involve explicit transfer of values. We shall see that this lifting is not needed in the register-based machine since the side channels via pop or swap do not exist in the register machine.

We now define what it means for a method to be typeable. In the definition below, we abbreviate $S(i)$ as S_i to simplify presentation.

Definition 2 (Typable method[5]). *A method m is typable w.r.t. a policy sgn and a CDR $\mathbf{region_m} : \mathcal{PP} \to \wp(\mathcal{PP})$ if there exists a security environment $se : \mathcal{PP} \to \mathcal{S}$ and a function $S : \mathcal{PP} \to \mathcal{S}^*$ s.t. $S_1 = \epsilon$ and for all $i, j \in \mathcal{PP}$:*

(a) $i \mapsto j$ implies there exists $st \in \mathcal{S}^*$ such that **region**$, se, sgn, i \vdash S_i \Rightarrow st$ *and $st \sqsubseteq S_j$;*
(b) $i \mapsto$ implies **region**$, se, sgn, i \vdash S_i \Rightarrow$

where $\sqsubseteq$ denotes the point-wise partial order on type stack w.r.t. the partial order taken on security levels.

The main result to prove w.r.t. the typing system above is its soundness, i.e., typeable programs are non-interferring. The definition of non-interference in this context depends on the notion of indistinguishability, from the view point of an observer, of various values associated with a program. The observer is associated with a security level, k_{obs}, denoting the upperbound of the security levels of objects and values it can observe. For the fragment of JVM we consider here, we only need to define a notion of variable indistinguishability.

Definition 3 (Local variables indistinguishability [5]). *Given the security levels $\vec{k_a}$ of local variables, the security level k_{obs} of observer, and $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$, we have $\rho \sim_{k_{\text{obs}}, \vec{k_a}} \rho'$ if ρ and ρ' have the same domain and $\rho(x) = \rho'(x)$ for all $x \in \mathbf{dom}(\rho)$ such that $\vec{k_a}(x) \leq k_{\text{obs}}$.*

Definition 4 (Non-interferent JVM method [5]). *A method m is* non-interferent *w.r.t. a policy $\vec{k_a} \to k_r$ and an attacker level k_{obs} if either $k_r \not\leq k_{\text{obs}}$, or for every ρ_1, ρ_2, r_1, r_2 such that $\langle 1, \rho_1, \epsilon\rangle \rightsquigarrow^+ r_1$, $\langle 1, \rho_2, \epsilon\rangle \rightsquigarrow^+ r_2$ and $\rho_1 \sim_{k_{\text{obs}}, \vec{k_a}} \rho_2$, we have $r_1 = r_2$.*

Theorem 1. *[5] Let P be a JVM typable program w.r.t. a safe CDR (***region**$, $**jun**$)$. Then P is non-interferent.*

2.2 DEX Type System

As is the case with our discussion on JVM in Section 2.1, we consider here only the set of DEX instructions needed for arithmetic instructions, given in Figure 4. For the extended set of instructions, please refer to Figure 4 in the section IV of the extended paper [18]. Unlike JVM, DEX does not use operand stacks, but uses virtual registers to perform similar roles as the operand stacks. So stack related

binop	$op\ r, r_a, r_b$	$\rho(r) = \rho(r_a)\ op\ \rho(r_b)$, a binary operation op on the values in r_a and r_b
const	r, v	$\rho(r) = v$, pushing a constant value v on register r
move	r, r_s	$\rho(r) = \rho(r_s)$, copy the value of register r_s to register r
ifeq	r, t	conditional jump if $\rho(r) = 0$
goto	t	unconditional jump
return	r_s	return the value of $\rho(r_s)$

where $op \in \{+, -, \times, /\}, v \in \mathbb{Z}, \{r, r_a, r_b, r_s\} \in \mathcal{R}, t \in \mathcal{PP}, c \in \mathcal{C}, f \in \mathcal{F}, m \in \mathcal{M}$, and $\rho : \mathcal{R} \to \mathbb{Z}$.

Fig. 4: DEX Instruction List

$$\frac{P[i] = \mathbf{const}(r, v) \quad r \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \ \rightsquigarrow\ \langle i+1, \rho \oplus \{r \mapsto v\}, h\rangle} \qquad \frac{P[i] = \mathbf{ifeq}(r, j) \quad r \in \mathbf{dom}(\rho) \quad \rho(r) = 0}{\langle i, \rho, h\rangle \ \rightsquigarrow\ \langle t, \rho, h\rangle}$$

$$\frac{P[i] = \mathbf{goto}(t)}{\langle i, \rho, h\rangle \ \rightsquigarrow\ \langle t, \rho, h\rangle} \qquad \frac{P[i] = \mathbf{ifeq}(r, t) \quad r \in \mathbf{dom}(\rho) \quad \rho(r) \neq 0}{\langle i, \rho, h\rangle \ \rightsquigarrow\ \langle i+1, \rho, h\rangle}$$

$$\frac{P[i] = \mathbf{return}(r_s) \quad r_s \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \ \rightsquigarrow\ \rho(r_s), h} \qquad \frac{P[i] = \mathbf{move}(r, r_s) \quad r, r_s \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \ \rightsquigarrow\ \langle i+1, \rho \oplus \{r \mapsto \rho(r_s)\}, h\rangle}$$

Fig. 5: DEX Operational Semantic (Selected)

instructions such as pop, push and swap are not present in DEX instructions. We denote with $\mathcal{R}$ the set of DEX virtual registers. In DEX bytecode, local variables are allocated virtual registers upon initialization of a method. So we shall not treat variables explicitly and only consider their proxies in the virtual registers. In the following, we assume that each local variable x is associated with a unique register r_x.

A state in DEX is a pair $\langle i, \rho\rangle$ consisting of a program counter i and a function ρ mapping registers to values. Figure 5 shows a selected set of rules of the operational semantics for DEX instructions. Please refer to [18] for the full list of DEX operational semantics.

The transfer rules of DEX are defined in terms of registers typing $rt : (\mathcal{R} \rightharpoonup \mathcal{S})$ instead of stack typing. Some of the transfer rules for DEX instructions are contained in Figure 6. Full transfer rules are contained in Figure 12 in the Appendix D of the extended paper.

The typability of the DEX closely follows that of the JVM, except that the relation between program points is $i \vdash RT_i \Rightarrow rt, rt \sqsubseteq RT_j$, where $RT : \mathcal{PP} \to rt$. We shall see later how we can construct a safe CDR for DEX from a safe CDR in JVM.

Theorem 2. *Let P be a DEX typable program w.r.t. a safe CDR (***region**, **jun***). Then P is non-interferent.*

3 The translation from JVM to DEX

We now describe the translation process from JVM to DVM. This is an abstracted version of what is implemented in the dx tool of Android. The dx tool

$$\frac{P[i] = \mathbf{const}(r, v)}{i \vdash rt \Rightarrow rt \oplus \{r \mapsto se(i)\}} \qquad \frac{P[i] = \mathbf{binop}(op, r, r_a, r_b)}{i \vdash rt \Rightarrow rt \oplus \{r \mapsto \big(rt(r_a) \sqcup rt(r_b) \sqcup se(i)\big)\}}$$

$$\frac{P[i] = \mathbf{move}(r, r_s)}{i \vdash rt \Rightarrow rt \oplus \{r \mapsto \big(rt(r_s) \sqcup se(i)\big)\}} \qquad \frac{P[i] = \mathbf{return}(r_s) \quad se(i) \sqcup rt(r_s) \leq k_r}{i \vdash rt \Rightarrow}$$

$$\frac{P[i] = \mathbf{ifeq}(r, t) \quad \forall_{j'} \in \mathbf{region}(i), se(i) \sqcup sec(r) \leq se(j')}{i \vdash rt \Rightarrow rt} \qquad \frac{P[i] = \mathbf{goto}(j)}{i \vdash rt \Rightarrow rt}$$

Fig. 6: DEX Transfer Rule (Selected)

push v	$\mapsto$	const(r_{ts}, v)
pop	$\mapsto$	$\varnothing$
swap	$\mapsto$	move(r_{ts+1}, r_{ts-1}), move(r_{ts}, r_{ts-2}), move(r_{t-1}, r_{ts}), move(r_{ts-2}, r_{ts+1})
load x	$\mapsto$	move(r_{ts}, x)
store x	$\mapsto$	move(r_x, r_{ts-1})
goto l	$\mapsto$	$\varnothing$
ifeq l	$\mapsto$	ifeq(r_{ts-1}, l)
return	$\mapsto$	move(r_0, r_{ts-1}) and (return(r_0) or goto(ret))

Fig. 7: JVM to DEX instruction translation (selected)

translates JVM in blocks of codes. To formalize this, it is useful to first define *Basic Block*, which is a construct containing a group of codes that has one entry point and one exit point (not necessarily one successor/one parent), has a parent list, a successor list, a primary succesor, and its order in the output phase.

Since DEX is register-based whereas JVM is stack-based, to bridge this gap, DEX uses registers to simulate JVM stacks. This is done as follows:

- We set aside l number of registers to hold local variables (register $0, \ldots, l$). We denote these registers with *locR*. For non-static methods, register r_0 always holds the value of the self-reference for the method (the variable *this*).
- A stack of size s is simulated by registers $l + 1, \ldots, l + s$.

Note that we assume the JVM bytecode has passed the Java bytecode verifier, which ensures, among others, that the maximum height of the operand stack in a method is fixed. Similarly, bytecode verifier guarantees that the (Java) types of the operand stack (and by implication, also its height) at each program point is fixed. This makes it possible to statically map each operand stack location to a register in DEX.

There are several phases to translate JVM bytecode into DEX bytecode (please refer to Table I the Section VI in [18] for the details of each step):

StartBlock: This phase determines the program point at which the instruction starts a block, and then creates a new block for each of these program points and associates it with a new empty block. A program point is a start of a block if it fulfills one of these conditions:

- It is the first instruction in a method;
- It is an instruction after a branching instruction (**ifeq**);
- it is a target of a branching instruction (**goto** and **ifeq**);

TraceParentChild: This resolves the parent and the successor relationship between blocks. Implicit in this phase is a step creating a temporary return block used to hold successors of the block containing return instructions. At this point, we assume that there is a special label called *ret* to address this temporary return block. The creation of a temporary return block depends on whether the function returns a value. If it is return void, then this block contains only the instruction **return-void**. Otherwise depending on the type returned (integer, wide, object, etc), the instruction is translated into the corresponding **move** and **return**. The **move** instruction moves the value from the register simulating the top of the stack to register r_0. Then **return** will just return the value of register r_0.

Translate: This phase translates each JVM instruction into a sequence (possibly zero) of DEX instructions. For the full translation scheme, refer to [18]. We outline some cases in Figure 7. In the figure, we assume that the index of the top of the operand stack is ts.

The **goto** instruction in JVM is not translated directly to **goto** in DEX. Rather, it is used to compute the relationship between blocks, from which new **goto** statements may be added later in the compilation process.

PickOrder: This phase orders blocks according to a "trace analysis" procedure. The trace analysis itself is quite simple in essence. That is, for each block we assign an integer denoting the order of appearance of that particular block. Starting from the initial block, we first assign an order to it. Then we pick the first unordered successor, giving priority to its primary successor (if any). The tracing continues until there is no more successor. After we reach one end, we pick an unordered block and do the trace analysis again. But this time we trace its source ancestor first, by tracing an unordered parent block and stop when there is no more unordered parent block or already forming a loop.

Output: This phase outputs the translated instructions based on the order of the blocks computed in the previous phase. During this phase, **goto** will be added for each block whose next block to output is not its successor. There is a special case for branching instruction (**ifeq**). If the next block to output is in fact the target of branching instead of its primary successor, then the branching instruction will be replaced by its opposite (**ifneq**). After the compiler has output all blocks, it will then read the list of DEX instructions and fix up the targets of jump instructions.

Definition 5 (Translated JVM Program). *The translation of a JVM program P into blocks whose JVM instructions are translated into DEX instructions is denoted by $[\![P]\!]$, where*

$$[\![P]\!] = \textbf{Translate}(\textbf{TraceParentChild}(\textbf{StartBlock}(P))).$$

Definition 6 (Output Translated Program). *The output of the translated JVM program $\llbracket P \rrbracket$ in which the blocks are ordered and then output into DEX program is denoted by $\lceil\llbracket P \rrbracket\rceil$, where $\lceil\llbracket P \rrbracket\rceil = \mathbf{Output}(\mathbf{PickOrder}(\llbracket P \rrbracket))$.*

Definition 7 (Compiled JVM Program). *The compilation of a JVM program P is denoted by $[\![P]\!]$, where $[\![P]\!] = \lceil\llbracket P \rrbracket\rceil$.*

4 Proof that Translation Preserves Typability

The first step to translating typing derivations from JVM to DEX is to translate the CDR from JVM to a CDR in DEX. The essential idea to the CDR-preservation property is the observation that if a program point i is in a region, then all program points in the basic block containing i belong to the same region. The only non-trivial property to prove is then to show that the goto statements between blocks link correct regions together. We state the CDR preservaton lemma here and defer the details for this translation to [18].

Lemma 1 (SOAP Preservation). *The SOAP properties are preserved in the translation from JVM to DEX, i.e. if the JVM program satisfies the SOAP properties, so does the translated DEX program.*

There are several assumption we make for this compilation. Firstly, the JVM program will not modify its self reference for an object. Secondly, since now we are going to work in blocks, the notion of se, S, and RT will also be defined in term of this addressing. A new scheme for addressing $blockAddress$ is defined from sets of pairs (bi, j), $bi \in blockIndex$, a set of all block indices (label of the first instruction in the block), where $\forall i \in \mathcal{PP}.\exists bi, j.$ s.t.$bi + j = i$. We also add additional relation $\Rightarrow^*$ to denote the reflexive and transitive closure of $\Rightarrow$ to simplify the typing relation between blocks.

We overload $\llbracket . \rrbracket$ and $[\![.]\!]$ to also apply to stack type to denote translation from stack type into typing for registers. This translation basically just maps each element of the stack to registers at the end of registers containing the local variables (with the top of the stack with larger index, i.e. stack expanding to the right).

More formally, suppose the maximum stack height allowed in a given method is M, the current program point is i, and suppose there are n local variables, denoted by $x_1, \ldots, x_n$, and that the height of the operand stack st at the current program point is k. We assume that the translated method in DEX has enough registers to simulate the operand stack. Then

$$\begin{aligned}[\![st]\!] = \{&r_0 \mapsto \vec{k_a}(x_1), \ldots, r_{n-1} \mapsto \vec{k_a}(x_n), r_n \mapsto st[0], \ldots, \\ &r_{n+k-1} \mapsto st[k-1], r_{n+k} \mapsto H, \ldots, r_{n+M} \mapsto H\}.\end{aligned}$$

At this point, it is important to note that since the stack height at a given program point i may be less than the maximum height allowed, we need to also define the security levels of registers that do not correspond to any variables or

operand stack element at program point i. These unused registers need to be given security level H (the top element of the security lattice) to ensure that the correct ordering of register types in the DEX type derivation. Intuitively, a stack operation such as pop or store consumes the top of the stack, but there is no corresponding elimination of the register corresponding to that stack element; registers cannot be 'popped' out of existence. By assigning the maximum security level to the unused registers, one also prevents implicit information flow through unused (or previously used) registers.

Lastly, the function $\lceil\!\lceil . \rceil\!\rceil$ is also overloaded for addressing (bi, i) to denote abstract address in the DEX side which will actually be instantiated when producing the output DEX program from the blocks.

Definition 8 (Stack Type Translation). $\forall i \in \mathcal{PP}, RT_{[\![i]\!][0]} = [\![S_i]\!]$.

The idea of the proof that compilation from JVM bytecode to DEX bytecode preserves typability is that any instruction that does not modify the block structure can be proved using Lemma 2 and Lemma 3 to prove the typability of register typing.

Lemma 2 (Typeable Sequence). *For any JVM program P with instruction Ins at address i , let the length of $[\![Ins]\!]$ be n. Let $RT_{[\![i]\!][0]} = [\![S_i]\!]$. If according to the transfer rule for $P[i] = Ins$ there exists st such that $i \vdash S_i \Rightarrow st$ then*

$$\begin{array}{c} \big(\ \forall 0 \le j < (n-1).\exists rt'.[\![i]\!][j] \vdash RT_{[\![i]\!][j]} \Rightarrow rt', rt' \sqsubseteq RT_{[\![i]\!][j+1]}\ \big) \\ \textit{and} \qquad \exists rt.[\![i]\!][n-1] \vdash RT_{[\![i]\!][n-1]} \Rightarrow rt, rt \sqsubseteq [\![st]\!] \end{array}$$

according to the typing rule(s) of $[\![Ins]\!]$.

Lemma 3 (Typeable Translation). *Let Ins be an instruction at address i, $i \mapsto j$, st, S_i and S_j are stack types such that $i \vdash S_i \Rightarrow st, st \sqsubseteq S_j$. Let n be the length of $[\![Ins]\!]$. Let $RT_{[\![i]\!][0]} = [\![S_i]\!]$, let $RT_{[\![j]\!][0]} = [\![S_j]\!]$ and rt be registers typing obtained from the transfer rules involved in $[\![Ins]\!]$. Then $rt \sqsubseteq RT_{[\![j]\!][0]}$.*

Lemma 4 (Constraint Satisfaction). *Let Ins be an instruction at program point i, S_i its corresponding stack types, and let $RT_{[\![i]\!][0]} = [\![S_i]\!]$. If $P[i]$ satisfy the typing constraint for Ins with the stack type S_i, then $\forall (bj, j) \in [\![i]\!].P_{DEX}[bj, j]$ will also satisfy the typing constraints for all instructions in $[\![Ins]\!]$ with the initial registers typing $RT_{[\![i]\!][0]}$.*

Using the above lemmas, we can prove that translated blocks are typable.

Lemma 5 (Translation Soundness). *Let P be a JVM program, if $\forall i, j.i \mapsto j.\exists st.i \vdash S_i \Rightarrow st$ and $st \sqsubseteq S_j$, then $[\![P]\!]$ will satisfy*

1. *for all blocks bi, bj s.t. $bi \mapsto bj$, $\exists rt_b$. s.t. $RTs_{bi} \Rightarrow^* rt_b, rt_b \sqsubseteq RTs_{bj}$; and*
2. *$\forall bi, i, j \in bi$. s.t. $(bi, i) \mapsto (bi, j).\exists rt$. s.t. $(bi, i) \vdash RT_{(bi,i)} \Rightarrow rt, rt \sqsubseteq RT_{(bi,j)}$*

where $RTs_{bi} = [\![S_i]\!]$ *with* $[\![i]\!] = (bi, 0)$, $RTs_{bj} = [\![S_j]\!]$ *with* $[\![j]\!] = (bj, 0)$,
$RT_{(bi,i)} = [\![S_{i'}]\!]$ *with* $[\![i']\!] = (bi, i)$, $RT_{(bi,j)} = [\![S_{j'}]\!]$ *with* $[\![j']\!] = (bj, j)$.

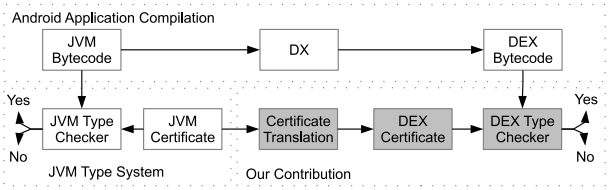

Fig. 8: Overall architecture

After we established that the translation into DEX instructions in the form of blocks preserves typability, we also need to ensure that the next phases in the translation process also preserves typability. The next phases are ordering the blocks, output the DEX code, then fix the branching targets.

Lemma 6 (Order and Output Soundness). *Let* $\lfloor\!\lfloor P \rfloor\!\rfloor$ *be typable basic blocks resulting from translation of JVM instructions still in the block form, i.e.*

$$\lfloor\!\lfloor P \rfloor\!\rfloor = \mathbf{Translate}(\mathbf{TraceParentChild}(\mathbf{StartBlock}(P))).$$

Given the ordering scheme to output the block contained in **PickOrder**, *then the output* $[\![P]\!]$ *is also typable.*

Finally, the main result of this paper in that the compilation of typable JVM bytecode will yield typable DEX bytecode which can be proved from Lemma 5 and Lemma 6. Typable DEX bytecode will also have the non-interferent property because it is based on a safe CDR (Lemma 1) according to DEX.

Theorem 3 (Compilation Soundness). *Let* P *be a typable JVM bytecode according to its safe CDR (*$\mathbf{region}, \mathbf{jun}$*), and method policies* Γ*, then* $[\![P]\!]$ *according to the translation scheme has the property that* $\forall i, j \in PP_{\mathrm{DEX}}$. *if* $i \mapsto j$. *then* $\exists rt. RT_i \Rightarrow rt, rt \sqsubseteq RT_j$ *according to a safe CDR (*$[\![\mathbf{region}]\!], [\![\mathbf{jun}]\!]$*).*

5 Implementation

Figure 8 shows the overall architecture of our work. Our contributions are highlighted in grey in the figure. As we have mentioned before, the Android compilation process starts from Java source and ends with into one or more DEX files, usually named "classes.dex". The rest of the toolchain will also bundle the manifest file and other resources into an APK. This APK can be installed into a mobile phone running Android OS as an app.

Our contributions are mainly contained in two components. The first component, certificate translation, takes the certificates for the source Java classes and translate it into a certificate for the corresponding DEX file, independently of the non-optimizing compilation done by the DX tool. The other component parse the DEX file, take the certificate, and check whether they match.

Certificate Structure Here we will describe how we structure our certificate, both for JVM and DEX. They mainly differ in that JVM uses stack type while

DEX uses registers type. Figure 9 shows high level structure of the certificate. Please refer Section VII.B of the extended paper for the details of how we represent each of the structure.

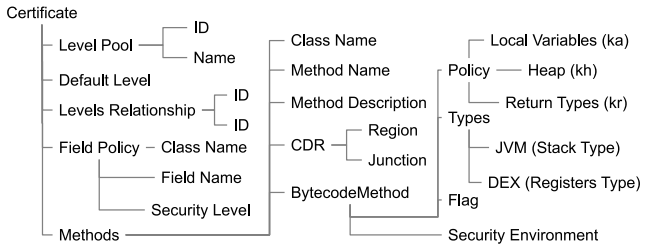

Fig. 9: Certificate Structure

Naive JVM Type Inference This component takes as an input a file which contains the certificate without the typing for each instructions (stacktype and security environment) then reconstruct the certificate for the JVM bytecode. The inference itself is quite simple: we just repeatedly infer the stack type and security environment for the successor instruction, starting from label 0, until they converge (no more change either in stack type or security environment).

For the type inference, we assign the instructions with least restrictive stack types and security level. Before we invoke the type inference algorithm, we first do a simple program flow tracing. The purpose of this program flow tracing is to ease the burden of type inference so that it just need to traverse the order produced by the flow tracer without having to deal with the case where an instruction still has not been assigned a stack type and security environment.

We implemented this in OCaml, in conjunction with the component to do a certificate translation. Our main consideration for doing so is because it is much easier to transfer the resulting JVM certificate directly to the next component within the program itself, rather than having a temporary output file.

Non-Optimizing Certificate Translation The translation component implements the translation process in Section 3. There are some additional complexity for the details of the implementation itself when we translate the type:

- The translation steps only concerns a specific bytecode under a specific policy, so we have to extend that to include each security level for the object that contains this method, the methods in the class, and we also have to extend the scope to include all the classes to be compiled to DEX bytecode.
- The convention for the class identifier is different between JVM and DEX. Simply put, a class identifier in DEX is "L" + JVM class identifier + ";", e.g. "java/lang/Object" in JVM becomes "Ljava/lang/Object;" in DEX. For the current implementation we also use the short method descriptor in DEX as opposed to the full method descriptor.

- The instruction pair of **Invoke** and **MoveResult** in DEX use the pseudo-register *ret*, which obviously is not captured in JVM. For **MoveResult**, firstly we take the stack type of the successor of **Invoke** then translate it into the registers type. After that we remove one value from the top of the stack and put it into *ret*. **Goto** instruction is also a tricky bit because it only ever appear in the translation phase if the next block to output is not a successor. To handle this, we maintain the registers type for the start of each block, and assign them to any **Goto** instruction that points to that block.
- To translate regions and junction we need to know the relation between the source program points and their translation (one to many mapping). The simplest part of the region translation is that for a program point $i = source(j)$, $\forall source(k) \in region(i), k \in region(j)$. Again, **MoveResult** and **Goto** complicate this process. For **MoveResult**, because it does not have its corresponding instruction in JVM we have to fix it so that it has the same source as the **Invoke** instruction in JVM. Similar case can be made with the **Goto** instruction in that the source instruction for **Goto** is the same as the start of the block which becomes the target of this **Goto** instruction. This decision of relating the Goto instruction with the target instead of the previous instruction is mainly to handle the case where the **Goto** instruction is generated by a branching instruction, in which case we need to include **Goto** in the region of the branching instruction as well.
- the flag that indicates whether register zero has been used or not does not exist in JVM. The translation itself just assigns the flags of all instructions to be zero, except the **Return** instruction and its associated **Move** and **Goto** instructions A difficulty arises in that we also need to take care of the moving of parameters which occurs in the first few instructions of the bytecode which also has a **Move** instruction targetting register zero. Fortunately this always happens as the first instruction in the bytecode, so we can ignore the rule about the flag if it is the first instruction in the bytecode .
- In JVM, all **Binop** instructions are using the values on the stack as the operands, regardless whether they are constants or not. In DEX, **Binop-Const** is dedicated to deal with a binary operation with constant, which has different size than regular **Binop** Instruction.
- The DEX translation of **Dup** is also not straightforward. Instead of copying the value from top of the stack and push it on top, it copy the value on top of the stack, create a register containing the copy, and then copy the value from this register to both the top of the stack and new top of the stack.

DEX Type Checker This last component do a simple type checking on the Android phone. It takes a DEX file, parse the DEX file, and check it against the certificate generated by the certificate translator. This means that our system works independently of the Android OS, and as such there is no modification at all to the overall Android structure. We implemented this as an Android application which takes an input string from user indicating the package name for the application that the user wants to type check. Several notes need to be made for this component:

– The DEX file in the package is contained in the predefined file name "classes.dex". We are aware of the possibility that there are multiple DEX file for a single application (e.g. when there are so many methods that it can not be contained within one DEX file), but it is not the main focus of our work.
– We also take the certificate from a predefined file name "Certificate.cert" contained in the "assets" directory. The type checker will just check for this particular file for the certificate.
– There are a lot of generated additional methods and classes for Android application, e.g. "Build.config" and "R" (and its subclasses). We decided to ignore them because they are not the focus of this work.
– Similar thing can be said about the methods and classes contained in the Android library. Although for this particular case, we decided to inject the certificate with these methods except that they are stripped off their bytecode instructions. This decision is mainly due to the transfer rule of method invocation which requires the policy of the target method.
– Parsing the whole DEX file takes a significant amount of time

The details of the algorithms can be seen in the extended version of this paper, available from the URL mentioned in the introduction.

6 Conclusion and Future Work

We presented the design of a type system for DEX programs and showed that the non-optimizing compilation done by the dx tool preserves the typability of JVM bytecode. Furthermore, the typability of the DEX program also implies its non-interference. We provide a proof-of-concept implementation illustrating the feasibility of the idea. This opens up the possibility of reusing analysis techniques applicable to Java bytecode for Android. As an immediate next step for this research, we plan to also take into account the optimization done in the dx tool to see whether typability is still preserved by the translation.

We currently have not modelled the information flow properties of various Android APIs, which are used pervasively in real-world applications. This is a subject of our on-going investigations.

References

1. DEX bytecode instructions. `http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`, accessed: 2014-12-31
2. Stat counter global stats. `http://gs.statcounter.com/#mobile_os-ww-monthly-201311-201411`, accessed: 2014-12-31
3. Appel, A.W.: Foundational proof-carrying code. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 247–256. IEEE Computer Society (2001), `http://dx.doi.org/10.1109/LICS.2001.932501`
4. Barthe, G., Naumann, D., Rezk, T.: Deriving an information flow checker and certifying compiler for java. In: Security and Privacy, 2006 IEEE Symposium on. pp. 13–pp. IEEE (2006)

5. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference java bytecode verifier. Mathematical Structures in Computer Science 23, 1032–1081 (10 2013), http://journals.cambridge.org/article_S0960129512000850
6. Barthe, G., Rezk, T., Saabas, A.: Proof obligations preserving compilation. In: Formal Aspects in Security and Trust, pp. 112–126. Springer (2006)
7. Bian, G., Nakayama, K., Kobayashi, Y., Maekawa, M.: Java bytecode dependence analysis for secure information flow. IJ Network Security 4(1), 59–68 (2007)
8. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a c compiler front-end. In: FM 2006: Formal Methods, pp. 460–475. Springer (2006)
9. Bugliesi, M., Calzavara, S., Spanò, A.: Lintent: towards security type-checking of android applications. In: Formal Techniques for Distributed Systems, pp. 289–304. Springer (2013)
10. Chaudhuri, A.: Language-based security on android. In: Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security. pp. 1–7. ACM (2009)
11. Davis, B., Beatty, A., Casey, K., Gregg, D., Waldron, J.: The case for virtual register machines. In: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators. pp. 41–49. IVME '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/858570.858575
12. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. Communications of the ACM 57(3), 99–106 (2014)
13. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 235–245. ACM (2009)
14. Enck, W., Ongtang, M., McDaniel, P.D., et al.: Understanding android security. IEEE security & privacy 7(1), 50–57 (2009)
15. Felt, A.P., Wang, H., Moschuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: 20th USENIX Security Symposium (2011)
16. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android's permission system. In: Computer Security—ESORICS 2012: 17th European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 7459, pp. 1–18 (Sep 2012), http://www.ece.cmu.edu/~lbauer/papers/2012/esorics2012-android.pdf
17. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications. Manuscript, Univ. of Maryland, http://www. cs. umd. edu/~ avik/projects/scandroidascaa (2009)
18. Gunadi, H., Tiu, A., Gore, R.: Formal certification of android bytecode. arXiv preprint arXiv:1504.01842 (2015)
19. Jia, L., Aljuraidan, J., Fragkaki, E., Bauer, L., Stroucken, M., Fukushima, K., Kiyomoto, S., Miyake, Y.: Run-time enforcement of information-flow properties on Android (extended abstract). In: Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security. pp. 775–792. Springer (Sep 2013), http://www.ece.cmu.edu/\~lbauer/papers/2013/esorics2013-android.pdf
20. Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: Static analyzer for detecting privacy leaks in android applications. MoST (2012)
21. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. ACM SIGPLAN Notices 41(1), 42–54 (2006)
22. Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., Weber, A.: Cassandra: Towards a certifying app store for android. In: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS

2014, Scottsdale, AZ, USA, November 03 - 07, 2014. pp. 93–104. ACM (2014), http://doi.acm.org/10.1145/2666620.2666631
23. Lortz, S., Mantel, H., Starostin, A., Weber, A.: A sound information-flow analysis for Cassandra. Tech. rep., TU Darmstadt (2014), technical Report TUD-CS-2014-0064
24. Necula, G.C.: Proof-carrying code. In: Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997. pp. 106–119. ACM Press (1997), http://doi.acm.org/10.1145/263699.263712
25. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Information systems security, pp. 1–25. Springer (2008)
26. Zhao, Z., Osono, F.C.C.: TrustDroid: Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In: Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on. pp. 135–143. IEEE (2012)