Cassandra: Towards a Certifying App Store for Android

Steffen Lortz Heiko Mantel Artem Starostin
Timo Bähr David Schneider Alexandra Weber
Computer Science Department, TU Darmstadt, Germany
<lastname>@mais.informatik.tu-darmstadt.de

ABSTRACT

Modern mobile devices store and process an abundance of data. Although many users consider some of this data as private, they do not yet obtain satisfactory support for controlling what applications might do with their data.

In this article, we propose Cassandra, a tool that enables users of mobile devices to check whether Android apps comply with their personal privacy requirements before installing these apps. Beyond this, Cassandra implements the core functionality of a conventional app store, including the browsing of available apps and the delivery of apps for installation. Cassandra performs the security analysis of apps on a server. However, a user does not need to trust this server because Cassandra employs the proof-carrying code paradigm such that the server's analysis result can be validated on the client. We have proven that Cassandra's security analysis soundly detects all potential information leaks, i.e., all flows of information that violate a user's privacy policy.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, correctness proofs*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*; D.4.6 [**Operating Systems**]: Security and Protection—*information flow controls*

Keywords

software security; mobility; proof-carrying code

1. INTRODUCTION

Cassandra enables a user to check Android apps against her personal security requirements before installation. The primary goal of Cassandra is to ensure that no private data and no other secrets are leaked by running an app. The silent leakage of a user's data is not just a technical possibility, but a serious threat in reality. For instance [10, 17] show that many apps forward private data to the Internet. Cassandra aims at countering this threat.

The overall functionality of Cassandra resembles the core functionality of conventional app stores like, e.g., F-Droid [12]. In particular, a user can browse the available apps on a server and can select apps for installation on her mobile device. The additional functionality offered by Cassandra is that a user can define security policies and verify that apps adhere to these policies before installation.

More specifically, a user-defined security policy specifies which flows of information are permissible and which ones are forbidden. Cassandra's security analysis checks such security policies against the flows of information that can be caused by running a given app. The security analysis is type based, and we have proven that it soundly enforces noninterference [16, 29] for a given security policy. Thus, if Cassandra's security analysis accepts an app, then it is guaranteed that this app's output to public sinks is independent from private information, where the security policy determines which sinks and sources of information are considered public and private, respectively. If the security analysis rejects an app, then Cassandra provides the user with detailed information about the possible violations of her security policy, enabling the user to make an informed decision whether to install the app, nevertheless, or not.

Android already provides security mechanisms. Before installing an app, a user needs to grant the app access permissions declared in the app's manifest. Android's permission system ensures that apps can only access protected resources of the mobile device if this is declared in their manifest. The permission system provides access control, but it does not control how information obtained from protected resources is propagated after a legitimate access. Android uses cryptographic signatures to ensure the authenticity and integrity of apps. These signatures are also used to constrain sharing of resources between apps, i.e., across their sandboxes. Android's built-in malware detection service verifies apps prior to installation and warns users about potential malware [2]. All these mechanisms are complementary to the information-flow analysis provided by Cassandra.

Starting with SCanDroid [13], a number of security analysis tools has been proposed for Android in recent years. For instance, AndroidLeaks [14], DidFail [19], IccTA [21], LeakMiner [31], Scandal [18], StaticChecker [23], and TrustDroid [32] support the static security analysis of Android apps at the bytecode level. All of these tools perform a data flow analysis and do not take implicit information flows properly into account. In contrast, FlowDroid [3] and the

SPSM'14, November 7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2666620.2666631.

Information Flow Checker [11] take implicit flows into account in their information flow analysis of Android apps at the bytecode level and source code level, respectively. However, there is no soundness result for these analyses. To our knowledge, Cassandra is the first tool for verifying the information flow security of Android apps that takes explicit as well as implicit information flows into account and whose security analysis comes with a soundness result. Moreover, Cassandra does not require any changes to the run-time environment as some other tools do like, e.g., TaintDroid [9].

The architecture of Cassandra builds on both the client-server paradigm and the proof-carrying code principle [24]. The client of Cassandra is an Android app itself that runs on off-the-shelf Android devices. Using the client, a user can browse available apps, specify security policies, initiate an information-flow analysis, and examine the analysis results. The server of Cassandra is implemented in Java and PHP. It runs on a conventional web server. Cassandra's server incorporates a database of available apps that can be browsed by a client. The server also performs the information-flow analysis for a given user-defined security policy. The results of a successful security analysis are communicated to the client in a security certificate that carries enough information to replay the security analysis on the client. This is where proof-carrying code is used to avoid that the server becomes part of the trusted computing base.

In this article, we present Cassandra, our prototypical certifying app store for Android. In particular, we

- demonstrate how Cassandra can be used to analyze the security of apps before installation on mobile devices,
- present Cassandra's architecture and implementation,
- describe the security type system underlying Cassandra's security analysis,
- define the noninterference-like security property that this security type system enforces, and present our soundness result showing that this is, indeed, the case.

A demo video showing the usage of Cassandra, a hands-on demo, and Cassandra's source code are available on-line.[1]

The structure of this article is as follows: Section 2 provides background on information-flow security and information leakage. Section 3 describes the usage of Cassandra. Section 4 presents the architecture of Cassandra and its implementation. Section 5 describes the theoretical foundation of Cassandra's information-flow analysis. The article concludes with a discussion of related work in Section 6 and a summary of our results and possible future work in Section 7.

Notational conventions. Given a set T, we use T^* to denote the set of all finite lists over T. We use [] to denote the empty list and $[t_0, \ldots, t_n]$ to denote the list with the elements $t_0, \ldots, t_n \in T$. Given a list $L \in T^*$, we use $\mathsf{length}(L)$ to denote the length of L and $L[i]$ to denote the i-th element of L, where $L[0]$ is the head of a nonempty list L. Given a set X, we use $\mathcal{P}(X)$ to denote the powerset of X.

2. INFORMATION FLOW SECURITY

An *information-flow policy* (brief: *flow policy*) defines a set of security domains $\mathcal{D}$, a domain assignment da, and an interference relation $\sqsubseteq \subseteq \mathcal{D} \times \mathcal{D}$. Each *security domain* (brief: *domain*) represents an abstract level of confidentiality. A *domain assignment* associates each concrete information source and sink in a program with such an abstract security domain. An *interference relation* is a partial order on domains that defines between which domains information may flow. A flow policy, hence, defines the permitted flows between sources and sinks in a program.

The semantics of information-flow policies is usually formalized using declarative properties in the spirit of *Noninterference* [16]. Noninterference-like properties require that if a given flow policy forbids information flow from a domain d to a domain d' (i.e. $d \not\sqsubseteq d'$) then a given program's output to sinks associated with level d' must be completely independent from input on sources associated with level d.

For violations of noninterference, one usually distinguishes between *explicit flows* and *implicit flows* of information. For example, consider the following two programs, where the variables output and phoneNo are of type natural number:

```
output = phoneNo;
```

```
output = 0;
while (0 < phoneNo) {
   phoneNo = phoneNo - 1;
   output = output + 1;
}
```

The program on the left explicitly assigns the value of phoneNo to output. Such an assignment is called an *explicit flow* of information. In contrast, the value of phoneNo is not assigned to output in the program on the right. Instead, the variable phoneNo influences the control flow and, depending on the control flow, the value of output is modified. Such a flow is called an *implicit flow* of information. In this example, the effects of the implicit flow and the explicit flow are the same: Both copy the initial value of phoneNo to output.

Throughout this article, we use a flow policy with two security domains $\mathcal{D} = \{low, high\}$ and the interference relation $\sqsubseteq = \{(low, low), (low, high), (high, high)\}$. The interference relation $\sqsubseteq$ specifies that information may flow between any two domains, except for from *high* to *low*. This is suitable for specifying that private information should not become public. To this end, one defines a domain assignment that associates all sources of private information and trusted sinks with the domain high, and all sources of public information and untrusted sinks with the domain low. Since the domain assignment depends on which sources and sinks are used in a program, we do not fix it here.

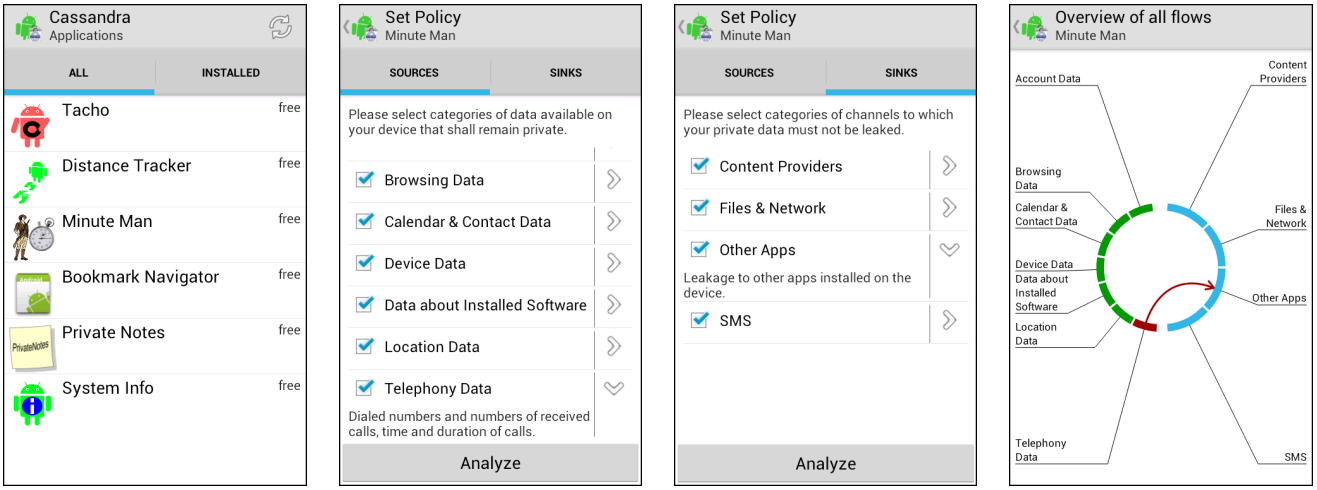
For instance, to express that no information should flow from phoneNo to output in the above example programs, one would choose a domain assignment that maps the variable output to the domain *low* and the variable phoneNo to the domain *high*. Under the resulting flow policy, both example programs violate noninterference, because the final value of output depends on the initial value of phoneNo.

3. USING CASSANDRA

The user of a mobile device can employ Cassandra to ensure that she only installs apps on the device that respect her security requirements. Installing apps using Cassandra is similar to installing apps from regular app stores like, e.g., F-Droid [12], but involves additional steps to choose a security policy and to analyze apps against this policy.

We illustrate the installation of apps with Cassandra at the example of the app Minute Man, an app for optimizing call costs. This app automatically interrupts phone calls after 59 seconds to avoid the charge for a second minute. We implemented Minute Man ourselves, inspired by similar existing Android apps for cost control, e.g., [28].

[1]URL of the project web-page of Cassandra: `http://www.mais.informatik.tu-darmstadt.de/cassandra.html`



(a) App Browser (b) Policy Editor (Sources) (c) Policy Editor (Sinks) (d) Reporter (Overview)

Figure 1: User interface of Cassandra's client

In the first step, the user invokes Cassandra's client app and then selects Minute Man among the available apps in Cassandra's *app browser*. Figure 1(a) shows a screenshot of the app browser's user interface. Afterwards, the user specifies which data is private and which sinks are untrusted by marking categories of information sources and sinks in Cassandra's *policy editor*. Our example user wants an overview of all information flows caused by running Minute Man. Hence, she marks all displayed categories of information sources and sinks as private and public, respectively. Figures 1(b) and 1(c) show screenshots of these selections.

In the second step, the user initiates the information-flow analysis by clicking the button "Analyze" (see bottom of Figure 1(b) and Figure 1(c)). The analysis itself is not performed on the client but on a server. After the security analysis is complete, the user can inspect the analysis result using Cassandra's *reporter*. The reporter either confirms that the user's security requirements are satisfied by the given app or provides information about all violations. In our example, the reporter warns about one violation, namely that telephony data might be leaked to other apps (see the arrow in Figure 1(d)). Based on the result of the security analysis, the user can make an informed decision whether to install the app or not.

In the last step, the user clicks the button "Install", given that she decided to install the app. This opens the standard installer of Android, which requests the permissions required by the app's manifest. If the user grants the permissions, the app is installed on the mobile device.

The user's choice of categories in the policy editor induces a domain assignment for an interface of an Android app. Abstractly, an Android app consists of fields and methods. Some of these methods are *entry points*, i.e., methods that can be called to run the app. Examples of entry points are Activity.onCreate(Bundle), which is called when an app is initially started, and LocationListener.onLocationChanged(Location), which is called when the mobile device's GPS location changes. The parameters of entry points are information sources in an app, their return values are information sinks. An app might also incorporate methods that read from or write to resources outside the app. Examples of such methods are TelephonyManager.getDeviceID(), which returns the unique identifier of the mobile device, and Context.startActivity(Intent), which starts a new activity given an intent. The return values of such methods are information sources in an app, whereas their parameters are information sinks.

By selecting a source category in the policy editor, the user indicates that all possible sources of information corresponding to this category shall be associated with *high*. By selecting a sink category, the user indicates that all sinks in this category shall be associated with *low*. All remaining sources and sinks in a given app are assigned the security domains *low* and *high*, respectively. The mapping of categories to sources and sinks is a one-to-many mapping, based on [23]. In our example analysis of Minute Man, the selection of the source category "Telephony Data" causes the domain assignment to assign *high*, e.g., to the return value of the method Intent.getStringExtra(Intent.EXTRA_PHONE_NUMBER). Moreover, the selection of the sink category "Other Apps" causes the domain assignment to assign *low*, e.g., to the parameter of the method Context.startActivity(Intent).

The domain assignment induced by a user's choice of categories in the policy editor, together with the set of security domains $\mathcal{D}$ and the interference relation $\sqsubseteq$, as defined in Section 2, constitutes a *user-defined information-flow policy*. If Cassandra reports no leaks for a given information-flow policy, then it is guaranteed that running the app does not cause any information flow that violates this flow policy. Otherwise, Cassandra reports for each leak the categories corresponding to the source and the sink of the leak.

In our example analysis, Cassandra reported an information leak from "Telephony Data" to "Other Apps". Indeed, the app Minute Man leaks the dialed phone number to the web browser. The fragment of Minute Man's source code that is responsible for this leak is shown below.

```
private String phonenumber;
private Context context
public void onReceive(Context context, Intent intent) {
  this.phonenumber =
      intent.getStringExtra(Intent.EXTRA_PHONE_NUMBER);
  this.context = context; ...
}
```

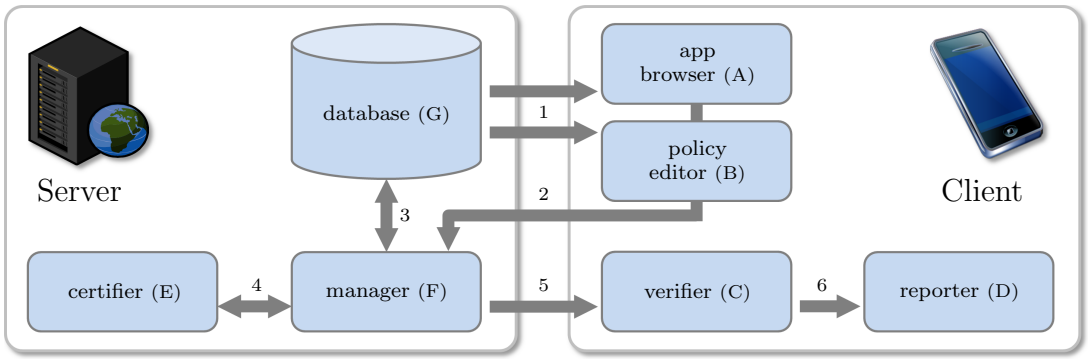


Figure 2: The components of Cassandra and their interaction

```
// the code below is executed after terminating a call
  long phoneNo = Long.parseLong(this.phonenumber);
  long output = 0;
  while (0 < phoneNo) {
      phoneNo = phoneNo - 1;
      output = output + 1;
  }
  String url="http://www.spp-rs3.de/cassandra/leak.php?nr=0X";
  url = url.replace("X", String.valueOf(output));
  try {
      Intent intent = Intent.parseUri(url, 0);
      intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
      this.context.startActivity(intent);
  } catch (URISyntaxException e) {}
```

When a phone call is made, the dialed phone number is retrieved by calling intent.getStringExtra(Intent.EXTRA_PHONE_NUMBER) and stored in the field phonenumber. When Minute Man terminates the call, the stored phone number is converted into a number stored in the variable phoneNo. The value of phoneNo is then implicitly copied to variable output by executing the while-loop. Afterwards, the character "X" in the URL is replaced by the copied number and an intent is created from the URL. Finally, the intent is used to start a new activity, i.e., the device's standard browser app, by calling context.start_Activity(intent). At this point, the information-flow policy from our example is violated because a dialed phone number has been leaked.

Note that the value of phoneNo implicitly flows to the variable output, i.e., the variable copied into the URL. Cassandra's ability to also detect implicit information flows is, hence, crucial for detecting the flow into the intent and, thus, the leak to another app. In its current implementation, Cassandra treats all intents, i.e., inter-app and intra-app intents, equally: An information flow into an intent that is used to start an activity, as in in the example of Minute Man, is always considered a potential flow to other apps.

4. DESIGN AND IMPLEMENTATION

In this section, we present the architecture of Cassandra and some technical details of Cassandra's implementation.

4.1 Architecture of Cassandra

The architecture of Cassandra builds on the client-server paradigm, where both the client and the server, again, have a modular architecture, as depicted in Figure 2.

Cassandra's client and server perform three interactions during the installation of an app. Firstly, the client retrieves the list of available apps and the corresponding list of categories of information sources and sinks from the server (1). Secondly, the client informs the server which app, which source categories, and which sink categories have been selected by the user (2). Thirdly, the server transmits the app and the result of the security analysis to the client (5). If an app violates the user-defined information-flow policy, then detailed information about each violation is sent to the client. If an app satisfies the policy, then a security certificate is sent that is based on the proof-carrying-code principle [24]. That is, the certificate contains enough information such that the client can efficiently replay the analysis.

Cassandra's server consists of three components. The *manager* (F) accepts requests by clients comprising the selection of an app, of source categories, and of sink categories. Given a request, the manager retrieves the app from the *database* (G) and then calls the *certifier* (E). The certifier computes the information flow policy induced by the selected categories and then performs an information-flow analysis for the app. Cassandra analyzes apps in a modular fashion. To enable this modularity, the certifier infers auxiliary information about all methods and fields of an app. If the app passes the analysis, then the certifier generates a security certificate and, otherwise, compiles a report of all security violations. The manager forwards the analysis results of the certifier to the client.

Cassandra's client consists of four components (see Figure 2). The app browser (A), the policy editor (B), and the reporter (D) together constitute the user interface of Cassandra's client app, which was already introduced in Section 3. The *verifier* (C) receives the security certificate generated by the server and replays the security analysis of the selected app. By this replay, Cassandra can ensure that it reports the satisfaction of a user-defined information-flow policy only if this policy is, indeed, satisfied. The replay is less expensive than the certification because the auxiliary information about methods and fields is contained in the security certificate and, hence, does not need to be recomputed.

Cassandra's use of proof-carrying code in the communication between server and client removes the server from the trusted computing base. That is, Cassandra offers security guarantees that are as reliable as if the analysis were performed by the client app, but without having to perform the full security analysis on the mobile device.

The modular architecture of Cassandra facilitates the development of alternative components. For instance, one could develop alternative policy editors or other information-flow analyses than our type-based analysis for Cassandra.

4.2 Security Certificates

Cassandra performs a type-based information-flow analysis that analyzes the methods of an app in a modular fashion. To this end, the analysis requires a typing of all fields and a typing of all methods in the app. As types, Cassandra uses security domains, i.e., elements of the set $\mathcal{D}$.

A *typing for fields* is a function in $\mathsf{FDA} = \mathcal{F} \to \mathcal{D}$ where $\mathcal{F}$ denotes the set of qualified field names. Fields are used to store information that should be accessible from different methods. The type of a field restricts the level of confidentiality up to which information may be stored in the field: If a field is typed *high*, then the field may be used to store private as well as public information. A field that is typed *low* may only be used for storing public information.

Methods are typed by method signatures. A *method signature* associates a type with each formal parameter and with the return value of a method. Let $\mathcal{M}$ denote the set of all qualified method names and the function $\mathsf{params} : \mathcal{M} \to \mathbb{N}_0$ specify the number of parameters of a method with a given name. For a method with name m and with n parameters, a method signature has the form

$$(m, [d_0, \ldots, d_{n-1}], ret) \in \mathcal{M} \times \mathcal{D}^* \times \mathcal{D} \ .$$

This method signature associates the method's return value with the type ret, the first parameter of the method with d_0, the second parameter with d_1, and so on. Intuitively, the method signature constitutes a contract that limits the permissible flows of information from a method's parameters to its return value: If the method m is called with arguments that are typed $d'_0, \ldots, d'_n$, where $d'_i \sqsubseteq d_i$ holds for each $i \in \{0, \ldots, n-1\}$, then the return value of the method may be assigned to sinks with domain d' only if $ret \sqsubseteq d'$ holds.

A *signature set* mda is a set of method signatures, i.e., $\mathsf{mda} \subseteq \mathcal{M} \times \mathcal{D}^* \times \mathcal{D}$ such that for all $(m, [d_0, \ldots, d_{n-1}], ret) \in \mathsf{mda}$ it holds that $\mathsf{params}(m) = i$. We denote the set of all such signature sets by MDA.

We say that a field typing in $\mathsf{fda} \in \mathsf{FDA}$ is *valid for an app* if and only if for each pair of fields $f_1, f_2 \in \mathcal{F}$ such that f_1 and f_2 have the same unqualified names and the class of f_1 inherits the field denoted by f_2, $\mathsf{fda}(f_1) = \mathsf{fda}(f_2)$ holds.

We say that a signature set $\mathsf{mda} \in \mathsf{MDA}$ is *valid for an app* if and only if for each pair of methods $m_1, m_2 \in \mathcal{M}$ such that m_1 and m_2 have the same unqualified names and m_1 denotes a method declared or inherited in a subclass of the class in which m_2 is declared, the signatures of m_2 cover the signatures of m_1, i.e.

$$\begin{array}{rl} & \{(m, [d_0, \ldots, d_{n-1}], ret) \in \mathsf{mda} \mid m = m_1\} \\ \subseteq & \{(m, [d_0, \ldots, d_{n-1}], ret) \in \mathsf{mda} \mid m = m_2\} \ . \end{array}$$

Note that a signature set $\mathsf{mda} \in \mathsf{MDA}$ associates method signatures also with methods that are entry points of an app and with methods that read and write to resources outside an app. Hence, the induced domain assignment da from Section 3 can be viewed as a partial signature set: Recall that a domain assignment da associates a parameter of an entry point with the security domain *high* if this parameter corresponds to a source category selected by the user in Cassandra's policy editor and with *low*, otherwise. Which security domains da assigns to return values of entry points and to arguments as well as return values of methods that access resources outside the app also depends on the user's choice of source and sink categories in the policy editor, as already explained in detail in Section 3.

We say that a signature set $\mathsf{mda} \in \mathsf{MDA}$ is *compatible with an app and a domain assignment* da if for each method $m \in \mathcal{M}$ that is an entry point of the app or that accesses resources outside the app exists one $(m', [d_0, \ldots, d_{n-1}], ret) \in \mathsf{mda}$ with $m' = m$, and for each $i \in \{0, \ldots, n-1\}$, d_i equals the domain that da assigns to the ith parameter of m, and ret equals the domain that da assigns to the return value of m.

A security certificate of an app comprises a valid typing for fields and a valid signature set for the app.

Definition 1. A *security certificate* for an app is a tuple $(\mathsf{fda}, \mathsf{mda}) \in \mathsf{FDA} \times \mathsf{MDA}$ such that fda and mda are valid for the app. A security certificate $(\mathsf{fda}, \mathsf{mda})$ for an app is *compatible with a flow policy* $(\mathcal{D}, \mathsf{da}, \sqsubseteq)$ if and only if mda is compatible with the app and da.

The certifier computes a compatible security certificate for an app and a user-defined flow policy in an incremental fashion, starting from a typing that associates all fields with *low* and an initial signature set $\underline{\mathsf{mda}} \in \mathsf{MDA}$ that captures the user's domain assignment. That is, $\underline{\mathsf{mda}}$ is a signature set that is compatible with the app and the user's domain assignment and that contains exactly one method signature for each entry point of the given app and for each method that accesses resources outside the app. Note that $\underline{\mathsf{mda}}$ is uniquely determined by a given app and domain assignment.

An *initial security certificate for an app and a domain assignment* is a pair $(\mathsf{fda}, \underline{\mathsf{mda}})$ such that fda is a field typing that associates all fields with *low* and $\underline{\mathsf{mda}}$ is the initial signature set for the given domain assignment.

4.3 Implementation of Cassandra

The client app of Cassandra is implemented in Java. Its manifest requires the permissions to access the Internet and to access the memory of the mobile device. These permissions are needed to communicate with the server and to store the apk-files of apps received from the server. The client app runs on unmodified Android devices, i.e., it requires no changes to the operating system or to the Dalvik VM. The server of Cassandra is implemented in Java and PHP. It is hosted on a conventional web-server with a MySQL database and a Java runtime environment.

The information-flow analysis that is performed by the certifier of Cassandra is implemented in Java. The implementation of the analysis is based on the security type system described in Section 5. It takes the bytecode extracted from the apk-file of an Android app as well as a security certificate as input and fully automatically computes a list of information flows in the app that violate the constraints imposed by the typings from the certificate.

The certifier repeatedly applies the information flow analysis to refine the security certificate in a stepwise manner, starting with an initial security certificate for the domain assignment induced by the user's selection of categories. In each iteration, the information-flow analysis either succeeds or reports security violations or the security certificate is refined. In particular, the certifier performs the following algorithm for a given app and user-defined flow policy $(\mathcal{D}, \sqsubseteq, \mathsf{da})$:

1. It computes an initial security certificate $(\mathsf{fda}, \underline{\mathsf{mda}})$ for the app and the domain assignment da.
2. It extends $\underline{\mathsf{mda}}$ to a signature set $\mathsf{mda} \in \mathsf{MDA}$ that is valid for the app, that includes all method signatures of $\underline{\mathsf{mda}}$, and that contains for each method that is not already covered by the method signatures in $\underline{\mathsf{mda}}$ a new

signature that types each parameter and the return value with the domain *low*.

3. It applies the information-flow analysis to the given app and the security certificate (fda, mda).
4. Where the analysis detects a flow of private information to a target (i.e., a field, a method parameter, or a return value) that is typed *low*, the certifier
   - records the information flow as a leak if the target is a sink that has been assigned *low* by da, or
   - adapts the typings such that they type the target with the security domain *high*, remain valid for the app, and compatible to da, otherwise.
5. The certifier repeats the process from step 3 until the typings and the leak records do not change any more.

Since an app has a finite number of fields and methods for which the typings can only be upgraded finitely often, the computation of the certificate is guaranteed to terminate.

If a security certificate is generated without recording an information leak, it is suitable for the verification of the app's information-flow security.

The current implementation of Cassandra's server supports the caching of analysis results in the database to increase efficiency. By checking whether a suitable security certificate already exists for an analysis request before calling the certifier, the manager avoids the unnecessary repetition of the certification algorithm.

Given a security certificate for an app, the verifier checks, firstly, that the typings in the certificate are valid for the selected app, secondly, that the certificate is compatible to the app and the flow policy induced by the user's choice of categories in the policy editor, and, thirdly, that the app adheres to the policy. The verification algorithm checks the adherence of an app to a flow policy with a single execution of the information-flow analysis without the need to repeat the iterative inference process conducted by the certification algorithm. In case the app passes the analysis with respect to the certificate, the verifier confirms the app's security to the reporter. In case the certificate is not suitable for the verification of the app's security, the analysis fails and the verifier reports an error.

5. THEORETICAL FOUNDATION

We formalize the type-based information flow analysis for a simplified, Dalvik-bytecode-like language, which we call ADL. ADL abstracts from details of the Dalvik bytecode language that are irrelevant for the flow of information in an app. Our abstraction reduces the number of instructions in our formalization and, hence, reduces conceptual complexity of the presentation and of our proofs. The mapping of Dalvik programs to ADL programs is straightforward.

After sketching the syntax and semantics of our language ADL in Sections 5.1 and 5.2, we present a noninterference-like condition that captures information-flow security in Section 5.3. We present our security type system at the level of ADL instructions in Section 5.4 and show that it soundly verifies the security condition. The mapping between the original Dalvik bytecode language and our language ADL is sketched in Section 5.5. The mapping shows that ADL covers almost the whole instruction set of the Dalvik bytecode language, 211 of 218 Dalvik bytecode instructions are covered. The mapping also provides a solid connection between the formalization of our security type system for ADL and its implementation for Dalvik bytecode in Cassandra.

Due to space restrictions, we cannot present all details about ADL, our security type system, our soundness proof, and the mapping to Dalvik bytecode in this article. These details are provided in a companion technical report [22].

5.1 Syntax of ADL

We denote the set of fully qualified class names with $\mathcal{C}$ and leave the set underspecified. The set $\mathcal{BOP}$ contains symbols for binary operations, such as addition. $\mathcal{ROP}$ is the set of symbols for operations that compare values, e.g., for equality tests. The set $\mathcal{N}$ contains numbers, e.g., integer numbers. The set $\mathcal{X}$ of register names is defined by $\mathcal{X} = \{x_i \mid i \in \mathbb{N}_0\}$. We define the ADL syntax based on these sets.

Definition 2. The set $\mathcal{I}$ of all *ADL instructions* contains

`binop` x_a, x_b, x_c, bop	`invoke-virtual-range` x_a, n, m
`new-instance` x_a, c	`if-test` x_a, x_b, n, rop
`iget` x_a, x_b, f	`return` x_a
`iput` x_a, x_b, f	

for arbitrary $x_a, x_b, x_c \in \mathcal{X}$, $n \in \mathcal{N}$, $bop \in \mathcal{BOP}$, $rop \in \mathcal{ROP}$, $c \in \mathcal{C}$, $f \in \mathcal{F}$, and $m \in \mathcal{M}$.

Methods comprise multiple instructions. We formalize methods by non-empty lists of instructions in $\mathcal{I}^* \setminus \{[]\}$.

The declaration of fields and methods in a program as well as the class hierarchy that specifies the inheritance of fields and methods are modeled by the partial functions $\mathsf{fdec} : \mathcal{F} \rightharpoonup \mathcal{F}$ and $\mathsf{mdec} : \mathcal{M} \times \mathcal{C} \rightharpoonup \mathcal{I}^* \setminus \{[]\}$. The function fdec is defined for all qualified field names $f, f' \in \mathcal{F}$ such that $\mathsf{fdec}(f) = f'$ if and only if f and f' have the same unqualified name and either $f = f'$ and the class of f declares the field, or the class of f inherits the field denoted by f'. The function mdec is defined for all $m \in \mathcal{M}$, $c \in \mathcal{C}$, and $M \in \mathcal{I}^* \setminus \{[]\}$ such that $M = \mathsf{mdec}(c, m)$ if and only if c is a subclass of or equal to the class of m, and M is either declared by c with the same unqualified name as m or inherited from the closest superclass of c that declares M with the same unqualified name as m. The class names, field names, and method names of a program are specified implicitly by the domains of fdec and mdec, the methods of the program by the range of mdec. The names of methods by which a program can be executed, the *entry points* of the program, are specified by a set $\mathsf{EP} \subseteq \mathcal{M}$.

Definition 3. An *ADL program* is a triple (EP, fdec, mdec) such that EP, dom(fdec), and dom(mdec) are finite.

An ADL program is *well-formed* if it satisfies the following properties: (1) Only class names, field names, and method names defined by the program are used in instructions. (2) Constants given as parameters of instructions are within sensible bounds (e.g., n in `if-test` n does not cause a jump outside the method). (3) The program is type-correct (e.g., a method is not invoked on a register containing a number). (4) Non-parameter registers of a method are written before they are read for the first time. (5) Each class has at most one immediate superclass. For the rest of this section, we assume only well-formed ADL programs.

5.2 Semantics of ADL

The operational semantics of ADL is defined in terms of a transition relation on states. A state specifies the position of the next instruction to be executed and the current content of the memory, consisting of registers and a heap.

We denote the set of locations by $\mathcal{L}$ and leave this set underspecified. The set $\mathcal{V} = \mathcal{N} \cup \mathcal{L}$ contains *values*, which can be stored in registers and fields. The register res is a distinguished register for storing the return values of methods. The elements of the set $\mathcal{R} = (\mathcal{X} \cup \{res\}) \to \mathcal{V}$ model *register states* as a mapping of register names to values. Each *object* from the set of objects $\mathcal{O} = \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$ consists of a class name specifying its type, and a mapping of its fields to values. A *heap* $h : \mathcal{L} \rightharpoonup \mathcal{O}$ models a memory for storing objects at locations. The set of heaps $\mathcal{H}$ is defined by $\mathcal{H} = \mathcal{L} \rightharpoonup \mathcal{O}$.

Definition 4. The set of *states* $\mathcal{S}$ is defined by $\mathcal{S} = \mathbb{N}_0 \times \mathcal{R} \times \mathcal{H}$. The set of *final states* $\mathcal{S}_{\mathsf{final}}$ is defined by $\mathcal{S}_{\mathsf{final}} = \mathcal{V} \times \mathcal{H}$.

States $\langle p, r, h\rangle \in \mathcal{S}$ consist of a program point p, a register state r, and a heap h. The program point p denotes the index of the next instruction to be executed in the current method. The register state and the heap model the current memory. When a method terminates, it yields a final state $\langle v, h\rangle \in \mathcal{S}_{\mathsf{final}}$, comprising the return value v of the method and the heap h at the time the method terminated.

The effect of executing an instruction in a given state is defined by the relation $\rightsquigarrow_{P,M} \subseteq \mathcal{S} \times (\mathcal{S} \cup \mathcal{S}_{\mathsf{final}})$. Intuitively, $\langle p, r, h\rangle \rightsquigarrow_{P,M} \langle p', r', h'\rangle$ models that the execution of the instruction at program point p of method M of program P with the register state r and the heap h results in the program point p', the register state r', and the heap h'. $\langle p, r, h\rangle \rightsquigarrow_{P,M} \langle v, h'\rangle$ models that the execution of the instruction at program point p with the register state r and the heap h terminates the execution of method M, returning the value v and updating the heap to h'. In the following, we provide intuitive semantics for the instructions from Definition 2 (for a formal definition of $\rightsquigarrow_{P,M}$, see [22]).

The instruction `binop` applies the binary operation specified by *bop* to the values of x_b and x_c, and stores the result in x_a. `new-instance` creates a null-initialized object of the class c on the heap and stores its location in x_a. `iget` copies the value of the field f of the object at the location given in x_b to x_a. `iput` copies the value of x_a to the field f of the object at the location given in x_b. The instruction`invoke-virtual-range` looks up the method with the name m in the class of the object at the location given in x_a. Then, this method is executed with the current heap and a fresh register state that maps the registers $x_0, \ldots, x_{\mathsf{params}(m)-1}$ to the values of $x_a, \ldots, x_{a+\mathsf{params}(m)-1}$. The returned value is stored in the distinguished register res. The instructions `binop`, `new-instance`, `iget`, `iput`, and `invoke-virtual-range` determine the next program point to be executed as their own program point plus one. The remaining instructions allow for nonlinear control flow: The instruction `if-test` yields the program point with the offset n from the current program point if the values in x_a and x_b are related with respect to *rop*. Otherwise, `if-test` yields its own program point plus one. The instruction `return` terminates the execution of the current method and returns the value of x_a.

The effect of executing an entire method M of a program P in a given state is defined by the relation $\Downarrow_{P,M}$.

Definition 5. Let $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ be a well-formed program and $M \in rng(\mathsf{mdec})$ be a method of P. The *execution relation* $\Downarrow_{P,M} \subseteq \mathcal{S} \times \mathcal{S}_{\mathsf{final}}$ is defined by the rules

$$\frac{s \rightsquigarrow_{P,M} t}{s \Downarrow_{P,M} t} \qquad \frac{s \rightsquigarrow_{P,M} s' \quad s' \Downarrow_{P,M} t}{s \Downarrow_{P,M} t}$$

where $s, s' \in \mathcal{S}$, and $t \in \mathcal{S}_{\mathsf{final}}$.

Intuitively, $\langle 0, r, h\rangle \Downarrow_{P,M} \langle v, h'\rangle$ denotes that the method M of program P executed in the initial register state r and heap h terminates in the heap h' and returns the result v.

The semantics of a program execution depends on the entry point used to start the program and on the class of the object on which the entry point is invoked. The semantics of the program is then defined by the semantics of the method that the program specifies for the given entry point and class.

Definition 6. Let $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ be a well-formed program, $m \in \mathsf{EP}$ be an entry point of P, and $c \in \mathcal{C}$ be a class name such that $(m, c) \in dom(\mathsf{mdec})$. The semantics of P with respect to m and c is defined by $\Downarrow_{P,\mathsf{mdec}(m,c)}$.

5.3 Security Property

This section introduces the declarative security property *TIN-ADL* (*Termination-Insensitive Noninterference for the Abstract Dalvik Language*). *TIN-ADL* intuitively requires that an observer who knows the content of public sources and sinks before and after the terminating execution of a program as well as the bytecode of the program does not learn anything about private information from executing the program. We assume that covert channels, e.g., runtime or power consumption of an app's execution, cannot be used to deduce private information. Which information sources and sinks of a program are public is specified by a security certificate as defined in Definition 1.

Although register states are considered to be not directly observable, registers are also typed with security domains to keep track of the confidentiality of their content. Typings for registers are functions from the set $\mathsf{RDA} = (\mathcal{X} \cup \{res\} \to \mathcal{D})$. The relation $\sqsubseteq$ on typings for registers is the pointwise extension of $\sqsubseteq$ on domains: For any two $\mathsf{rda}_1, \mathsf{rda}_2 \in \mathsf{RDA}$, $\mathsf{rda}_1 \sqsubseteq \mathsf{rda}_2$ holds if and only if $\mathsf{rda}_1(x) \sqsubseteq \mathsf{rda}_2(x)$ holds for all registers $x \in \mathcal{X} \cup \{res\}$.

Given typings to determine which parameters, return values, registers, and fields are potentially observable, the observer's capabilities to distinguish values, register states, and heaps can be formalized. To support a notion of indistinguishability that is independent from the actual placement of objects on the heap, we follow the approach of Banerjee and Naumann [4] and relate the locations of any two corresponding observable objects with a partial injective function on locations $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$.

Definition 7. Let $\mathsf{rda} \in \mathsf{RDA}$ be a typing for registers, $\mathsf{fda} \in \mathsf{FDA}$ be a typing for fields, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function on locations. The *indistinguishability* of any two values $v_1, v_2 \in \mathcal{V}$, register states $r_1, r_2 \in \mathcal{R}$, objects $o_1, o_2 \in \mathcal{O}$, and heaps $h_1, h_2 \in \mathcal{H}$ with respect to rda, fda, and β (written $v_1 \sim_\beta v_2$, $r_1 \sim_{\beta,\mathsf{rda}} r_2$, $o_1 \sim_{\beta,\mathsf{fda}} o_2$, and $h_1 \sim_{\beta,\mathsf{fda}} h_2$, respectively) is defined by the following rules:

$$\frac{n \in \mathcal{N} \quad v_1 = v_2 = n}{v_1 \sim_\beta v_2} \qquad \frac{v_1, v_2 \in \mathcal{L} \quad \beta(v_1) = v_2}{v_1 \sim_\beta v_2}$$

$$\frac{\forall x \in \mathcal{X} \cup \{res\}.(\mathsf{rda}(x) = low \implies r_1(x) \sim_\beta r_2(x))}{r_1 \sim_{\beta,\mathsf{rda}} r_2}$$

$$\frac{\begin{array}{c} o_1 = (c_1, F_1) \qquad o_2 = (c_2, F_2) \qquad c_1 = c_2 \\ \forall f \in dom(F_1).(\mathsf{fda}(f) = low \implies F_1(f) \sim_\beta F_2(f)) \end{array}}{o_1 \sim_{\beta,\mathsf{fda}} o_2}$$

$$\frac{\begin{array}{c} dom(\beta) \subseteq dom(h_1) \quad rng(\beta) \subseteq dom(h_2) \\ \forall \ell \in dom(\beta).h_1(\ell) \sim_{\beta,\mathsf{fda}} h_2(\beta(\ell)) \end{array}}{h_1 \sim_{\beta,\mathsf{fda}} h_2}$$

Two values are indistinguishable if they are the same numerical value or if they are the locations of corresponding objects. Two register states are indistinguishable if all registers with the type *low* hold indistinguishable values. Two objects are indistinguishable if they are instances of the same class and all fields with the type *low* hold indistinguishable values. Two heaps are indistinguishable if for all locations on the first heap that are potentially observable, there exists a corresponding location on the second heap such that the objects at both locations are indistinguishable.

The indistinguishability relations are the basis for formalizing *TIN-ADL*: If the two initial states of a method execution are indistinguishable to an observer, i.e., if the method is executed with indistinguishable heaps and parameters, then the final states after the method's execution must also be indistinguishable. Otherwise, the observer would learn about differences of private information in the initial states.

Definition 8. Let $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ be a well-formed program, $M \in rng(\mathsf{mdec})$ be a method of P, $\mathsf{fda} \in \mathsf{FDA}$ be a valid typing for the fields of P, $m \in \mathcal{M}$ be a method name, and $d_0, \ldots, d_{i-1}, ret \in \mathcal{D}$ for $i = \mathsf{params}(m)$ be domains.

The method M satisfies *TIN-ADL* with respect to fda and $(m, [d_0, \ldots, d_{i-1}], ret)$ if and only if there exists a typing for registers $\mathsf{rda} \in \mathsf{RDA}$ with $d_k \sqsubseteq \mathsf{rda}(x_k)$ for all $k \in \{0, \ldots, i-1\}$ and for all partial injective functions $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$, register states $r_1, r_2 \in \mathcal{R}$, heaps $h_1, h_2, h_1', h_2' \in \mathcal{H}$, and return values $v_1, v_2 \in \mathcal{V}$ such that

$$r_1 \sim_{\beta,\mathsf{rda}} r_2,$$
$$h_1 \sim_{\beta,\mathsf{fda}} h_2,$$
$$\langle 0, r_1, h_1\rangle \Downarrow_{P,M} \langle v_1, h_1'\rangle, \text{ and}$$
$$\langle 0, r_2, h_2\rangle \Downarrow_{P,M} \langle v_2, h_2'\rangle,$$

there exists a partial injective function on locations $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$, such that $\beta \subseteq \beta'$, $h_1' \sim_{\beta',\mathsf{fda}} h_2'$ and, if $ret = low$, $v_1 \sim_{\beta'} v_2$.

A method M satisfies *TIN-ADL* with respect to a valid typing for fields and a method signature if and only if for any two terminating executions of M from initial configurations with indistinguishable registers and heaps, the final configurations have indistinguishable heaps and, if the return value is public, the returned values are indistinguishable.

TIN-ADL for methods is lifted to programs by requiring each method corresponding to an entry point of the program to satisfy *TIN-ADL* for all signatures of the entry point.

Definition 9. Let $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ be a well-formed program, and $(\mathsf{fda}, \mathsf{mda})$ be a security certificate for P. The program P satisfies *TIN-ADL* with respect to $(\mathsf{fda}, \mathsf{mda})$ if and only if for all entry points $m \in \mathsf{EP}$, for all security domains $d_0, \ldots, d_i, ret \in \mathcal{D}$ such that $(m, [d_0, \ldots, d_i], ret) \in \mathsf{mda}$, and for all classes $c \in \mathcal{C}$ and methods $M \in rng(\mathsf{mdec})$ such that $M = \mathsf{mdec}(m, c)$, method M satisfies *TIN-ADL* with respect to fda and $(m, [d_0, \ldots, d_i], ret)$.

Intuitively, *TIN-ADL* requires that if any method corresponding to an entry point of the program is executed in any two initial states that are indistinguishable to an observer, then the two final states of the execution are also indistinguishable to the observer. Hence, the execution of a program that satisfies *TIN-ADL* never leaks information from private information sources to public sinks.

5.4 Security Type System

Verifying that a program satisfies *TIN-ADL* directly based on the program's semantics is a tedious task. This section introduces a security type system to facilitate the automatic verification that a given program satisfies *TIN-ADL*.

The security type system captures both explicit leaks of information and implicit leaks due to control-flow dependencies on private information. Since the syntax of ADL does not convey control-flow dependencies between program points, the control-flow dependencies have to be determined based on the instruction semantics. To make the control-flow dependencies between program points in a method explicit, the security type system uses the concept of security environments and control dependence regions, which we adopt from a security type system for Java bytecode by Barthe, Pichardie, and Rezk [5].

A *security environment* $se : \mathbb{N}_0 \to \mathcal{D}$ for a method specifies for each program point of the method an upper bound of the security domains of all information sources that determine whether the program point is executed or not.

On which other instructions in a method the execution of a given instruction depends we derive from the successor relation of the method. For any well-formed program $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ and method $M \in rng(\mathsf{mdec})$ of P, the *successor relation* $\rightarrow_{P,M} \subseteq \mathbb{N}_0 \times \mathbb{N}_0$ is defined by $\rightarrow_{P,M} = \{(p, p') \in \mathbb{N}_0 \times \mathbb{N}_0 \mid \exists r, r' \in \mathcal{R}, h, h' \in \mathcal{H}. \langle p, r, h\rangle \rightsquigarrow_{P,M} \langle p', r', h'\rangle\}$. Based on the successor relation, the control-flow dependencies between program points can be approximated such that the approximation is safe.

Definition 10. Let $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ be a well-formed program, and $M \in rng(\mathsf{mdec})$ be a method of P. The functions $region_{P,M} : \mathbb{N}_0 \to \mathcal{P}(\mathbb{N}_0)$ and $jun_{P,M} : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ are a safe over-approximation of M's control dependence regions if they satisfy the three safe over-approximation properties:

1. For all program points $p, p', p'' \in \mathbb{N}_0$ such that $p \rightarrow_{P,M} p'$, $p \rightarrow_{P,M} p''$, and $p' \neq p''$ (i.e., p is a branching point), $p' \in region_{P,M}(p)$ or $p' = jun_{P,M}(p)$.
2. For all program points $p, p', p'' \in \mathbb{N}_0$, if $p' \rightarrow_{P,M} p''$ and $p' \in region_{P,M}(p)$, then either $p'' \in region_{P,M}(p)$ or $p'' = jun_{P,M}(p)$.
3. For all program points $p, p' \in \mathbb{N}_0$, if $p' \in region_{P,M}(p)$ and there exists no $p'' \in \mathbb{N}_0$ such that $p' \rightarrow_{P,M} p''$, then $jun_{P,M}(p)$ is undefined.

For all branching points p, a control dependence region $region_{P,M}(p)$ that is a safe over-approximation of M's control flow contains at least those program points that are executed depending on what the branching condition of the instruction at p evaluates to. The junction point $jun_{P,M}(p)$ specifies the end of the control dependence region of program point p in the sense that it points to an instruction that is executed independently of the evaluation of the branching condition. If the method terminates in a control dependence region, the region cannot have a junction point.

The typing rules of the security type system are parametric in the method M, the control dependence region $region_{P,M}$, the typing for fields fda, the signature set mda, the domain $ret \in \mathcal{D}$ of the return value of M, and the security environment se. The judgment

$$M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \rightarrow \mathsf{rda}'$$

denotes that after executing the instruction at program point p in the context of $M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se$, the typ-

$$\frac{M[p] = \texttt{binop}\ x_a, x_b, x_c, bop \quad \mathsf{rda}' = \mathsf{rda}[x_a \mapsto \mathsf{rda}(x_b) \sqcup \mathsf{rda}(x_c) \sqcup se(p)]}{M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \to \mathsf{rda}'}$$

$$\frac{M[p] = \texttt{new-instance}\ x_a, c \quad \mathsf{rda}' = \mathsf{rda}[x_a \mapsto se(p)]}{M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \to \mathsf{rda}'}$$

$$\frac{M[p] = \texttt{iget}\ x_a, x_b, f \quad \mathsf{rda}' = \mathsf{rda}[x_a \mapsto \mathsf{rda}(x_b) \sqcup \mathsf{fda}(f) \sqcup se(p)]}{M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \to \mathsf{rda}'}$$

$$\frac{M[p] = \texttt{iput}\ x_a, x_b, f \quad \mathsf{rda}(x_a) \sqcup \mathsf{rda}(x_b) \sqcup se(p) \sqsubseteq \mathsf{fda}(f)}{M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \to \mathsf{rda}}$$

$$\frac{M[p] = \texttt{invoke-virtual-range}\ x_a, n, m \quad (m, [\mathsf{rda}(x_a), \ldots, \mathsf{rda}(x_{a+n-1})], ret') \in \mathsf{mda} \quad se(p) = low \quad \mathsf{rda}(x_a) = low \quad \mathsf{rda}' = \mathsf{rda}[res \mapsto ret']}{M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \to \mathsf{rda}'}$$

$$\frac{M[p] = \texttt{if-test}\ x_a, x_b, n, rop \quad \forall j \in region_{P,M}(p).\mathsf{rda}(x_a) \sqcup \mathsf{rda}(x_b) \sqsubseteq se(j)}{M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \to \mathsf{rda}}$$

$$\frac{M[p] = \texttt{return}\ x_a \quad se(p) \sqcup \mathsf{rda}(x_a) \sqsubseteq ret}{M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda} \to \mathsf{rda}}$$

Figure 3: Typing rules for the instructions from $\mathcal{I}$

ing for the registers must be rda' if it was rda before. A program point within a method is typable, if a judgment is derivable with respect to the typing rules in Figure 3. In the definition of these rules, we write $d_1 \sqcup d_2$ to denote the least upper bound of two domains $d_1, d_2 \in \mathcal{D}$ with respect to the interference relation $\sqsubseteq$.

The rule for `binop` types the target register with the least upper bound of the domains of the argument registers and of the security environment in which the instruction is executed. The rule for `new-instance` types the target register with the domain of the security environment. The typing rule for `iget` types the target register the least upper bound of the domains of the register holding the location of the object, the source field, and the security environment. The rule for `iput` ensures that the target field is typed with at least the highest of the domains of the register holding the location of the object, the source register, and the security environment. The rule for `invoke-virtual-range` ensures that methods are not executed depending on private information ($se(p) = low$) or on an object at a location from a private register ($\mathsf{rda}(x_a) = low$). Moreover, the rule ensures that a method signature exists in the signature set that types the parameters of the invoked method with the same domains as the typing of the registers containing the arguments. If such a signature exists, the result register is typed with the type of the return value declared by the signature. If multiple such signatures exist then the least restrictive one is chosen, i.e., the one which assigns the lowest security domain to the return value. The rule for `if-test` ensures that the security environment of all program points in the control dependence region of the program point is at least the upper bound of the domains of the registers in the branching condition. The rule for `return` ensures that the type of the return value ret of the current method is at least as high as the domain of the register that contains the return value and the domain of the security environment.

Taking care that the targets of assignments are typed with at least the highest domain of all information sources forbids explicit leaks of information. The requirement that the target registers, fields, and return values are typed with a domain at least as high as the current security environment rules out implicit information leaks due to control-flow dependencies on private information. The premise of the rule for `if-test` ensures that all control-flow dependencies are taken into account by the security environment. In case of field accesses, the target register is typed with a domain at least as high as the domain of the register holding the location of the object to prevent implicit information leaks due to aliasing. The requirement that methods are not executed depending on private information or on an object at a location from a private register prevents implicit leaks due to observable effects that executing the method may have on the heap.

An entire method is typable if there exists a declaration of the security environment and a typing of registers for each program point such that for each potential step in the method's execution, a suitable judgment can be derived.

Definition 11. Let $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ be a well-formed program, $M \in rng(\mathsf{mdec})$ be a method of P such that $\mathsf{length}(M) = i$ for some $i \in \mathbb{N}_0$, $(\mathsf{fda}, \mathsf{mda})$ be a security certificate for P, and $region_{P,M} : \mathbb{N}_0 \to \mathcal{P}(\mathbb{N}_0)$ be a safe overapproximation of M's control dependence regions. Furthermore, let $m \in \mathcal{M}$ be a method name, and $d_0, \ldots, d_j, ret \in \mathcal{D}$ be security domains such that $(m, [d_0, \ldots, d_j], ret) \in \mathsf{mda}$.

The method M is typable with respect to the signature $(m, [d_0, \ldots, d_j], ret)$, $(\mathsf{fda}, \mathsf{mda})$, and $region_{P,M}$ if and only if there exist a security environment $se : \mathbb{N}_0 \to \mathcal{D}$ and typings for registers $\mathsf{rda}_0, \ldots, \mathsf{rda}_{i-1} \in \mathsf{RDA}$ such that

1. for all $k \in \{0, \ldots, j\}$ it holds that $d_k \sqsubseteq \mathsf{rda}_0(x_k)$,
2. for all $p, p' \in \mathbb{N}_0$, if $p \to_{P,M} p'$, there exists a typing for registers $\mathsf{rda}' \in \mathsf{RDA}$ such that $\mathsf{rda}' \sqsubseteq \mathsf{rda}_{p'}$ and
$$M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda}_p \to \mathsf{rda}'$$
is derivable, and
3. for all $p \in \mathbb{N}_0$, if there exists no $p' \in \mathbb{N}_0$ such that $p \to_{P,M} p'$, then the judgment
$$M, region_{P,M}, \mathsf{fda}, \mathsf{mda}, ret, se \vdash p : \mathsf{rda}_p \to \mathsf{rda}_p$$
is derivable.

The first condition ensures that the method treats the parameters given in the initial register state at least as confidential as they have been declared by the method signature. The second condition requires that a typing rule is applicable for each possible transition between program points p and p' in the method such that the typing for registers resulting from the derivable judgment rda' does not classify more registers as private than the fixed typing for registers $\mathsf{rda}_{p'}$. The third condition requires that each return instruction in the method is typable.

The typability of methods is extended to ADL programs by requiring that each of the program's methods is typable with respect to all method signatures that could possibly refer to the respective method.

Definition 12. Let $P = (\mathsf{EP}, \mathsf{fdec}, \mathsf{mdec})$ be a well-formed program, and $(\mathsf{fda}, \mathsf{mda})$ be a security certificate for P. The program P is typable with respect to $(\mathsf{fda}, \mathsf{mda})$ if and only if for all names of methods $m \in \mathcal{M}$, classes $c \in \mathcal{C}$, methods $M \in rng(\mathsf{mdec})$, and security domains $d_0, \ldots, d_i, ret \in \mathcal{D}$ such that $M = \mathsf{mdec}(m, c)$ and $(m, [d_0, \ldots, d_i], ret) \in \mathsf{mda}$, there exists a safe over-approximation of M's control dependence regions $region_{P,M} : \mathbb{N}_0 \to \mathcal{P}(\mathbb{N}_0)$ such that the method M is typable with respect to $(m, [d_0, \ldots, d_i], ret)$, $(\mathsf{fda}, \mathsf{mda})$, and $region_{P,M}$.

We have shown in [22] that if a program is typable with respect to a security certificate, then it satisfies *TIN-ADL* with respect to the certificate.

THEOREM 1. *For all well-formed programs* $P = (\mathsf{EP}, \mathit{fdec}, \mathit{mdec})$*, and security certificates* $(\mathsf{fda}, \mathsf{mda})$ *for* P*, if program* P *is typable with respect to* $(\mathsf{fda}, \mathsf{mda})$*, then* P *satisfies TIN-ADL with respect to* $(\mathsf{fda}, \mathsf{mda})$*.*

5.5 From ADL to Dalvik Bytecode

We designed ADL with the goal to focus on aspects of the original Dalvik bytecode language that are relevant for analyzing the information flows in apps. Hence, ADL abstracts from Dalvik's details that have no impact on the flow of information, such as Dalvik bytecode instructions that exist in different variants for arguments of different types. For example, the Dalvik bytecode instructions `iget`, `iget-object`, `iget-boolean`, `iget-byte`, `iget-char`, and `iget-short` essentially all load a value from a field to a register. These instructions are subsumed by the single ADL instruction `iget`. Further language details of Dalvik bytecode from which ADL abstracts include the bit-width of constant arguments to some Dalvik bytecode instructions (e.g., `const`, `const/4`, `const/16`, `const/high16`), the concrete computation performed by instructions for unary and binary operations (e.g., `add-int`, `sub-int`, `mul-int`, and so on). The complete mapping of ADL instructions to their corresponding instructions in Dalvik bytecode is specified in [22].

Overall, the instruction set of ADL comprises 55 instructions that correspond to 211 Dalvik bytecode instructions. In particular, ADL covers all arithmetic instructions, all instructions for the creation and manipulation of objects and arrays, all instructions for variants of method calls, and all instructions that operate on 64 bit arguments (i.e., on arguments comprising two registers) in the Dalvik bytecode language. Moreover, ADL supports the Dalvik bytecode instructions for jumps and branching, except for `packed-switch` and `sparse-switch`. Two aspects of Dalvik bytecode are not yet supported in ADL are the handling of exceptions (i.e., the Dalvik bytecode instructions `check-cast`, `move-exception`, and `throw`) as well as synchronization (i.e., the instructions `monitor-enter` and `monitor-exit`).

Also note that our definition of well-formedness of ADL programs in Section 5.1 imposes conditions on ADL programs that are analogous to the conditions enforced by the Dalvik bytecode verifier on Dalvik bytecode programs [1].

In Section 5.4, we defined the rules of our security type system for ADL, but we implemented typing rules for Dalvik bytecode within Cassandra. The mapping of the typing rules for ADL to the implemented typing rules for Dalvik bytecode is straightforward. The typing rule for each Dalvik bytecode instruction in our implementation is analogous to our ADL typing rule for the corresponding ADL instruction. For example, the typing rule for the Dalvik instruction `iget-object` is analogous to the ADL typing rule for `iget`. Our implementation also supports the analysis of apps with instructions for the invocation of native methods and methods of the Android framework that are not automatically analyzable with our implementation. For such methods, trusted signatures may be specified manually. We have shown in [22] that our analysis is sound in the presence of trusted method signatures if the respective methods satisfy *TIN-ADL* with respect to these signatures. This condition could either be verified manually or with the help of other analysis tools.

Despite the abstraction, our ADL instructions capture all aspects of the corresponding Dalvik instructions that are relevant for the flow of information when running a program. Given this, our definition of the security property *TIN-ADL* carries over to Dalvik bytecode: The execution of an Android app consists of a sequence of executions of its entry points reacting to different events, e.g., start of the app, user inputs, and termination of the app. For an app that satisfies *TIN-ADL*, each entry point satisfies *TIN-ADL* and, hence, does not leak private information when executed. Thus, private information is disclosed at no point in any execution sequence of the app. We are confident that our soundness result also carries over, but we have not formally verified the soundness of Cassandra's implementation.

6. RELATED WORK

Related work for Cassandra includes research on language-based information-flow security and on Android application security. Sabelfeld and Myers [25] provide a broad overview of language-based security, whereas Enck [8] reviews the research on Android application security. In this section, we focus on concepts for security-certifying app stores, on tools for the information-flow analysis of Android apps, and on type-based information-flow analyses with proven soundness that are applicable for Android and Java programs.

App-store concepts. The approach closest to Cassandra is a prototypical extension of TouchDevelop [30], a platform for developing, distributing, and running apps on different operating systems for mobile devices. The extension supports the static detection of information flows in TouchDevelop apps and the limitation of these flows during runtime. When running a TouchDevelop app for the first time, the user gets an overview of potential information flows in the app and is asked whether and how these flows shall be mitigated. The information-flow analysis is based on abstract interpretation and detects explicit leaks and implicit leaks due to control-flow dependencies on private information. Yet, the information-flow analysis of TouchDevelop is limited to apps that have been developed with TouchDevelop and, thus, cannot be applied to Android apps.

Androlyzer [6] employs a collection of different security analysis tools to inspect Android apps for security problems. For each app, an overview of the analysis results, e.g., potential leaks of sensitive information off the device, is presented to the user, which allows to assess the risk of installing the app. An asset of Cassandra in comparison with this tool is that it provides explicit security guarantees in the case that no security problems are found.

Gilbert et al. [15], Titze et al. [27], Zhauniarovich et al. [33], and Ernst et al. [11] propose app stores employing (semi-)automatic security analyses to verify the security of

an app before making it available for installation. Thus, apps with malicious behavior are excluded from the store or at least reported to the user as potentially dangerous. Cassandra checks apps against user-defined information-flow policies rather than against security policies of the app store provider. Thus, Cassandra gives a user the possibility to specify and check individual flow policies for each app.

Fuchs et al. [13] suggested to employ proof-carrying-code with their security analysis tool SCanDroid for analyzing Android apps on a server and verifying the results on a mobile device. However, it was not elaborated how this could be achieved.

Tools for information-flow analysis. SCanDroid [13], AndroidLeaks [14], LeakMiner [31], ScanDal [18], StaticChecker [23], TrustDroid [32], DidFail [19], and IccTA [21] support the static detection of explicit data leaks in Android apps. Cassandra, in addition, considers implicit information flows through control-flow dependencies on secrets.

The tools FlowDroid [3] and Information Flow Checker [11] support the static analysis of Android apps at the level of Dalvik bytecode and Java source code, respectively. They detect both explicit and implicit information leaks. Both analysis tools were designed for precisely tracking the information flows within apps. In contrast, the focus when designing the analysis method of Cassandra was on achieving provable soundness rather than on maximizing precision.

This makes Cassandra, to the best of our knowledge, the first security analysis for Android apps that has been proven sound and detects explicit and implicit information leaks.

Dynamic approaches to verify the information-flow security of apps like TaintDroid [9] can precisely detect information leakage on mobile devices. To this end, they taint data and track its propagation within and across apps. Yet, dynamic information-flow analyses require modifications to the operating system or the instrumentation of the apps and induce runtime overhead by monitoring the propagation of data. By employing a static analysis method, Cassandra neither requires the modification of the analyzed apps nor of the operating system and, once installed, analyzed apps are executed without any runtime overhead.

Security type systems. The first security type system for an imperative high-level programming language equipped with a formal proof of soundness was proposed by Volpano, Irvine, and Smith [29]. Banerjee and Naumann [4] adopted this concept to define a sound security type system for programs written in a fragment of the JavaCard programming language. Security type systems for Java could be used to analyze Android apps, but only if the source code is available. For some apps, the source code even cannot be obtained using decompilers [10]. Our security type system was developed specifically to analyze Dalvik bytecode, such that access to the source code of an app or the decompilation of Dalvik binaries are not necessary.

The first security type system with proven soundness for a bytecode language was proposed by Kobayashi and Shirane [20]. They analyzed a subset of Java bytecode without object orientation. Barthe, Pichardie, and Rezk [5] provided a sound security type system for a larger subset of Java bytecode that includes objects, method calls, arrays, and exceptions. We adopted some aspects of the security type system from [5] when defining our security type system for Dalvik bytecode, e.g., the handling of implicit information flows in unstructured bytecode and the structure of the soundness proof. Yet, there exist nontrivial differences between Java bytecode and Dalvik bytecode that had to be considered. For example, Dalvik programs have multiple entry points while Java programs have a single main-method, and Dalvik bytecode operates on registers whereas the Java Virtual Machine uses an operand stack.

7. CONCLUSION

To our knowledge, Cassandra offers the first information-flow analysis for Android apps within a prototypical app store to enforce user-specified security requirements. Moreover, we are not aware of other information-flow analyses for Dalvik bytecode with a soundness result. Our analysis could also be used separately from Cassandra, e.g., as a third-party service allowing to upload and analyze apps, but we have not explored this possibility yet.

In this article, we used the app Minute Man to illustrate the use of Cassandra. Cassandra's database contains further apps whose security we have successfully analyzed. For instance, the *Distance Tracker* app measures a user's travel distance using the device's GPS location, and the *Private Notes* app allows a user to manage personal notes. Like for Minute Man, the functionality of these self-implemented apps is similar to that of existing apps (see, e.g., [26, 7]). In the future, the collection of apps that are available in Cassandra's database shall grow further. In this context, we plan to conduct an experimental evaluation of the performance and the precision of Cassandra's information-flow analysis with respect to third-party open-source apps.

We also plan to extend the coverage of the Dalvik bytecode language by Cassandra's information-flow analysis in both the implementation and the underlying theory. Cassandra already covers 211 of 218 Dalvik instructions. We intend to add support for the remaining instructions, in particular for exception handling and synchronization, in a stepwise manner. Adding these instructions will also be helpful for being able to analyze a wider range of functionalities provided by existing Android apps.

Acknowledgments. We thank Jan E. Keller for contributing to an early version of Cassandra and the anonymous reviewers for providing valuable comments. This work was supported by the DFG under the project RSCP (MA 3326/4-2) in the Priority Program RS^3, and by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE.

8. REFERENCES

[1] Android Open Source Project. Dalvik Bytecode Verifier Notes. `https://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/verifier.html`. Accessed in March 2014.

[2] Android Open Source Project. Security Enhancements in Android 4.2. `http://source.android.com/devices/tech/security/enhancements42.html`. Accessed in March 2014.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field,

Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[4] A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[5] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In *European Symposium on Programming*, pages 125–140, 2007.

[6] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities Within Android Applications. In *International Conference on Malicious and Unwanted Software*, pages 66–72, 2011.

[7] Crazelle Solutions. My Notes. `https://play.google.com/store/apps/details?id=com.crazelle.app.notepad`. Accessed in March 2014.

[8] W. Enck. Defending Users Against Smartphone Apps: Techniques and Future Directions. In *International Conference on Information Systems Security*, pages 49–70, 2011.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Conference on Operating Systems Design and Implementation*, pages 393–407, 2010.

[10] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011.

[11] M. D. Ernst, R. Just, S. Millstein, W. M. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative Verification of Information Flow for a High-Assurance App Store. Technical report, University of Washington, USA, 2014.

[12] F-Droid Ltd. F-Droid. `https://f-droid.org/`. Accessed in March 2014.

[13] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, University of Maryland, USA, 2009.

[14] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307, 2012.

[15] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *International Workshop on Mobile Cloud Computing and Services*, pages 21–26, 2011.

[16] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[17] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112, 2012.

[18] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Mobile Security Technologies*, 2012.

[19] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android Taint Flow Analysis for App Sets. In *ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.

[20] N. Kobayashi and K. Shirane. Type-Based Information Analysis for Low-Level Languages. In *Asian Workshop on Programming Languages and Systems*, pages 302–316, 2002.

[21] L. Li, A. Bartel, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. I Know What Leaked in Your Pocket: Uncovering Privacy Leaks on Android Apps with Static Taint Analysis. Technical report, University of Luxembourg, 2014.

[22] S. Lortz, H. Mantel, A. Starostin, and A. Weber. A Sound Information-Flow Analysis for Cassandra. Technical report, TU Darmstadt, Germany, 2014.

[23] C. Mann and A. Starostin. A Framework for Static Detection of Privacy Leaks in Android Applications. In *Symposium on Applied Computing*, pages 1457–1462, 2012.

[24] G. C. Necula. Proof-carrying Code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[25] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[26] Sports Tracking Technologies Ltd. Sports Tracker. `https://play.google.com/store/apps/details?id=com.stt.android`. Accessed in March 2014.

[27] D. Titze, P. Stephanow, and J. Schuette. App-Ray: User-driven and Fully Automated Android App Security Assessment. Technical report, Fraunhofer AISEC, Germany, 2013.

[28] TouchSpot MX. Call Timer. `https://play.google.com/store/apps/details?id=com.gary.NoTePases`. Accessed in March 2014.

[29] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[30] X. Xiao, N. Tillmann, M. Fahndrich, J. De Halleux, and M. Moskal. User-aware Privacy Control via Extended Static-information-flow Analysis. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 80–89, 2012.

[31] Z. Yang and M. Yang. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *Third World Congress on Software Engineering*, pages 101–104, 2012.

[32] Z. Zhao and F. C. Colón Osorio. "TrustDroid": Preventing the Use of SmartPhones for Information Leaking in Corporate Networks through the Used of Static Analysis Taint Tracking. In *International Conference On Malicious And Unwanted Software*, pages 135–143, 2012.

[33] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo. TruStore: Implementing a Trusted Store for Android. Technical report, University of Trento, Italy, 2014.