DROIDFORCE: Enforcing Complex, Data-Centric, System-Wide Policies in Android

Siegfried Rasthofer*, Steven Arzt*, Enrico Lovat†, Eric Bodden*

*Secure Software Engineering Group, EC SPRIDE, Technische Universität Darmstadt, {firstname.lastname}@ec-spride.de
†Technische Universität München, enrico.lovat@in.tum.de

***Abstract*—Smartphones are nowadays used to store and process many kinds of privacy-sensitive data such as contacts, photos, and e-mails. Sensors provide access to the phone's physical location, and can record audio and video. While this is convenient for many applications, it also makes smartphones a worthwhile target for attackers providing malicious applications. Current approaches to runtime enforcement try to mitigate unauthorized leaks of confidential data. However, they are often capable of enforcing only a very limited set of policies, like preventing data leaks only within single components or monitoring access only to specific sensitive system resources.**

In this work, we present DROIDFORCE, an approach for enforcing complex, data-centric, system-wide policies on Android applications. DROIDFORCE allows users to specify fine-grained constraints on how and when which data may be processed on their phones, regardless of whether the malicious behavior is distributed over different colluding components or even applications. Policies can be dynamically exchanged at runtime and no modifications to the operating system nor root access to the phone are required.

DROIDFORCE works purely on the application level. It provides a centralized policy decision point as a dedicated Android application and it instruments a decentralized policy enforcement point into every target application. Analyzing and instrumenting an application takes in total less than a minute and secured applications exhibit no noticeable slowdown in practice.

I. INTRODUCTION

Over the last years, Android became the most prevalent platform in the smartphone market, with a market share of more than 79% [1]. Applications are not only provided by Google as the operating system's vendor, but also by arbitrary independent developers. As of now, there are more than one million applications available through the official Google Play store alone and numerous other markets exist. While this abundance of applications that cater to almost every need is convenient for the user, it also makes security and privacy goals harder to achieve. In combination with the impact given by the large number of devices used across the globe, this makes modern smartphones a worthwhile target for attackers.

According to a recent security report of McAfee [2], one of the major security threats in Android applications stems from malicious applications stealing the user's money through premium-rate SMS messages. This kind of malware sends text messages to premium-rate numbers that cost up to 10$ per message [3]. Such behavior can easily be prevented by blocking premium-SMS messages, at the price, however, of disabling legitimate premium-rate services too. A more fine grained user-defined policy that, e.g., limits the number or the frequency of SMS messages that can be sent to a specific set of telephone numbers, is therefore advantageous. Yet, a simple per-application counter would not suffice: Since attackers try to avoid detection, they may spread the malicious behavior across multiple applications, each of which only sends a handful of messages. The individual applications may look benign to a user who is unaware of the hidden functionality. The abuse could only be detected (and prevented) by security mechanisms with a **system-wide** perspective, configured through a language that is **expressive** enough to e.g. specify the maximum number of messages that could be sent to a particular recipient within a certain amount of time[1].

Furthermore, modern smartphones store and process a large number of private data such as contacts, SMS messages, photos and e-mails. When installing an application, users can only review basic access-control policies in terms of permissions granted to the application, e.g., "allow access to location data". Still, the user has no control over how the application processes this data or whether it discloses it to untrusted parties. In this example, the location could, for instance, be used to track the user's movements and report them to a remote server [4].

Like in the premium-SMS example, this problem is aggravated in practical scenarios in which the user not only runs a single application in isolation, but dozens of them. While the data leaks of a single application might not be a large concern, combining data from several applications may allow for knowledge aggregation that severely threatens the privacy of the user. Consider a weather application sending out the user's current location and a ticket booking application requiring the user's full name. Both are acceptable behaviors, but an adversary running the shared remote server for both applications would be able to associate the user's movements with her identity, a significant threat to the user's privacy. Hence, in addition to the system-wide perspective, a security mechanism must be able to track which sensitive data is processed and disclosed by each application and prevent unwanted combinations, e.g. location data and full-name should not be sent to the same server. Dually, the policy language to express such requirements must allow the specification of requirements also in terms of data, so-called **data-centric** policies, rather than pure events.

The task is more complex than tracking how data flows from sources to sinks in a specific application, which in itself is

[1]Android 4.2 and later already implements a check on premium-rate SMS messages and asks the user for confirmation. This is however a non-customizable simple pattern matching, whereas our language allows complex temporal and global properties on various types of data, not only SMS numbers.

already non-trivial [5]. Android applications can be developed based on four components: activities for UI interaction, services for long-time running tasks, broadcast receivers for receiving broadcasts and content providers which manage the access to a structured set of data. Three of them - activities, broadcast receivers and services - can communicate with each other with asynchronous messages, so-called *intents*, also used for inter-app communication. For instance, the Facebook messenger app contains more than 300 different activities and more than 30 different services, all of which can potentially communicate with each other. This shows that an analysis of a single component within an application is not sufficient for data flow tracking, although most of the current data flow tracking tools [5]–[8] are able to track data only within a single component.

Consider a third example of an application designed to steal contacts from the user's address book and send them to a remote attacker. For not requiring a suspicious set of permissions (i.e., requesting access to both the Internet and the list of contacts), the functionality is split among multiple colluding applications. The first application only reads the contacts and sends them to the next one, which in turn propagates them to the next one, and so on, until the last application finally sends them to a remote server. Even worse, such collusion attacks do not always require all the colluding applications to be malware; one malicious application may leverage vulnerabilities or bugs in other already installed benign applications to use functionalities (like "accessing the Internet") for which the malicious application does not have the appropriate permissions [9]. Some applications also fail to properly secure access to their own data, another vulnerability that could be exploited by malicious applications installed on the same phone to extract sensitive information [6]. These kind of collusion attacks are omnipresent in real-world applications [10]. Regardless of the intention of the collusion, forbidding such behavior requires system-wide data flow tracking capabilities and an enforcement mechanism able to track and possibly ban undesired **inter-application data flows**.

Lastly, policies are not fixed. Privacy preferences of end-users differ, may change over time, and companies might need to enforce even more complex rules for compliance and intellectual property protection. It is therefore important for a security framework to allow policies to be **dynamically configured** on the user level without having to change the underlying detection and enforcement mechanisms. Additionally, such enforcement must be able to run on an **unmodified** stock version of the **Android operating system** as users cannot be expected to re-flash their device, thereby voiding its warranty, or to create additional security threats by rooting it. In corporate environments, the enforcement mechanism must be guaranteed not to interfere with additional security mechanisms that may already be in place such as a sandboxed or partitioned operating system [11]. Technically, this limits practically feasible approaches to the application level.

In this paper, we present DROIDFORCE, a tool for enforcing complex, data-centric, system-wide, policies on Android applications. DROIDFORCE works by instrumenting applications which can afterward run on unmodified stock version of Android operating system. No root access is required, because DROIDFORCE directly integrates the policy enforcement logic into the target applications and thus does not require any modifications to the Android framework or kernel. The policies enforced by DROIDFORCE are global, like "no more than 2 SMS messages per day may be sent to +01-234-56789, regardless of the sending application". Policies can refer to data being transmitted between applications in order to prevent the collusion attacks described above, like in the example of the weather-application, for which a suitable policy would be "No single server may receive both the user's location and full name (regardless of which application is actually sending it)".

We evaluate DROIDFORCE on a number of real-world Android applications and show that it is capable of enforcing practically relevant system-wide security policies without inducing notable overhead during normal application use. We also evaluate the security of DROIDFORCE's framework against tampering and policy circumvention attempts by malicious applications. We made our source code and evaluation results available as an open source project at https://github.com/secure-software-engineering/DroidForce

To summarize, the original contribution of this paper is the first approach for Android security that supports *all* of the following properties:

- **Global:** DROIDFORCE can enforce system-wide policies, i.e. constraints for a single application as well as for multiple applications at the same time;
- **Data-centric:** DROIDFORCE's policies can be expressed in terms of events and in terms of sensitive data, like location data, contacts data, or IMEI;
- **Expressive:** DROIDFORCE's policies are written in OSL [12], a first-order linear temporal logic language with support for cardinality and time constraints (see section IV);
- **Dynamic:** DROIDFORCE's policies can be deployed and revoked at runtime, without any change required to the installed applications;
- **Inter-application:** DROIDFORCE tracks flows of sensitive data through single application (statically) and across different components and different applications (dynamically);
- **Non-intrusiveness:** DROIDFORCE does not require any modification to the underlying Android operating system, nor rooting the device;
- **Practical:** DROIDFORCE's practical relevance is supported by tests with real-world applications, policies and attack scenarios.

The remainder of this paper is structured as follows: firstly, we discuss the details about the architecture (Section II) followed by details about the framework (Section III), analyzing in details each of the three examples presented above; then we introduce the policy specification language supported by DROIDFORCE (Section IV); afterward, in Section V we present the results of our performance evaluation and our security analysis; in Section VI we justify the relevance of our approach by comparing it to existing solutions from the literature and in Section VII we conclude and discuss future work.

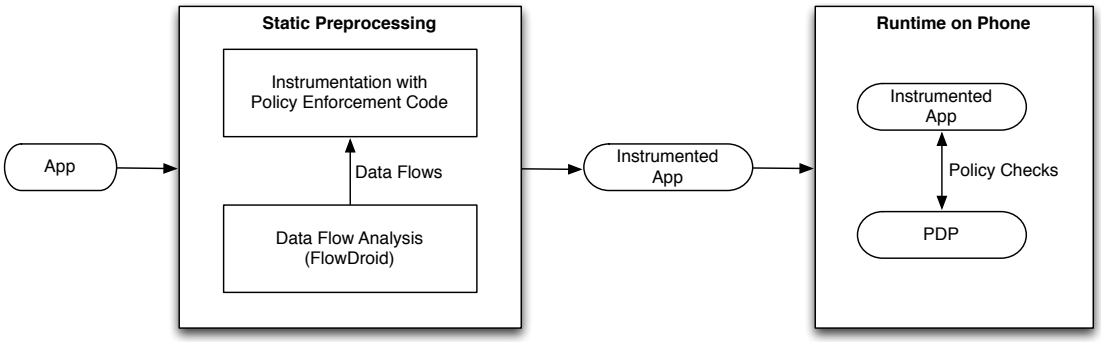

Fig. 1. DROIDFORCE Architecture Overview

II. THE ARCHITECTURE

DROIDFORCE consists of two main architectural components: The *Policy Decision Point* (PDP) and the *Policy Enforcement Point* (PEP). The PDP is a central independent entity responsible for storing and managing user-defined policies, like those described in Section IV. The PEP is in charge of intercepting system events and inhibiting their execution if allowing it would violate a user policy. In DROIDFORCE, the PEP is injected into each application using Soot [13], an open source bytecode instrumentation tool. The policy decision point is implemented as a single centralized Android application with which all PEPs interact (see Figure 2).

Whenever a protected operation like sending an SMS message is about to occur in an application, the enforcement point generates a respective PDP request, e.g. "Application A is about to send an SMS message with text X to phone number Y". All requests are transmitted to the PDP. Only if the PDP confirms that the event is allowed under the current policy, the PEP allows the respective application to continue. If the PDP denies the execution of the event, the protected operation is skipped, and the execution resumes from the next instruction in the application code. While denying the execution of some instructions might lead to unexpected crashes, most of the time the application can properly handle the denied request. As an example, consider an application disclosing the user's location for targeted-advertisement purposes. Inhibiting the http request containing the location implies that no response for the request is ever received. Many advertisement libraries are programmed very defensively, and in such a case, they simply do not display the advertisement (instead of crashing or blocking the whole application). The same defensive behavior is also implemented by background tasks, like uploading usage statistics, which can also be safely inhibited.

Executing sensitive operations such as sending SMS messages, or opening Internet connections are done via calls to API methods in Android applications. During the instrumentation phase, DROIDFORCE uses Soot to wrap these calls with policy checks that (1) generate a request for the PDP, (2) send it, and (3) only execute the API call if the response is positive (allow). Policies can optionally specify additional compensative actions to be taken in response to attempted violations, such as reporting the event or executing user-defined code. This feature is useful as it allows policies to take corrective measures like reporting targeted attacks on company smartphones to the IT security department or automatically blocking and uninstalling the respective applications via the company's mobile device management (MDM) solution. The code can either be executed in the context of the application itself, allowing for further inspection into the monitored application's state, or in the context of the PDP application, which is a secured environment.

Figure 1 shows an overview of DROIDFORCE's architecture. The PEP must be able to intercept every protected operation in all applications on the phone. DROIDFORCE therefore instruments a decentralized enforcement point into every user application during a preprocessing step on a desktop computer or server before the respective application is deployed onto the device. On this server, DROIDFORCE first analyzes the application for data flows using the FlowDroid [5] data flow tracker. FlowDroid is a context-, flow-, field-, object-sensitive and lifecycle-aware data flow tracking tool, and is one of the most precise data flow trackers currently available [5]. The statically extracted dependencies between sources and sinks are instrumented into the application in form of a table, alongside the code responsible for policy enforcement. After the instrumented application is installed on the user phone, the enforcement code can interact with the PDP at runtime to check whether protected operations shall be allowed or inhibited. The data flow table is queried to obtain the origin of the data being transmitted across components or applications, in order to generate proper PDP requests.

III. THE FRAMEWORK

This section elaborates on DROIDFORCE's architecture presented in Section II using the examples given in Section I and is structured as follows: Section III-A describes the enforcement of a simple policy against the abuse of costly premium SMS messages. Section III-B shows how data aggregation attacks can be prevented using DROIDFORCE. Section III-C discusses active collusion attacks between multiple applications which attempt to stealthily leak privacy-sensitive data. The advantages and drawbacks of the design decisions in DROIDFORCE's architecture are discussed in Section III-D.

A. Example 1: Premium SMS Messages

In this example, a malicious application tries to steal money from the user by sending a large number of SMS messages to costly premium-rate numbers, which is explicitly forbidden by a user policy. Per telephone number, only two premium-rate messages may be sent in a day. Note that this policy is not an artificial one. Malware sending premium-rate SMS messages indeed introduce artificial delays after each sending in order not to be detected (e.g., Android.FakeRegSMS.B [14]).

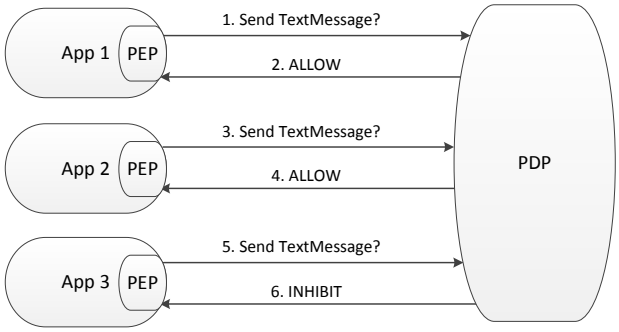

Fig. 2. Enforcement of the global policy "no more than 2 SMS per day can be sent", assuming the last request happens within 24 hours after the first.

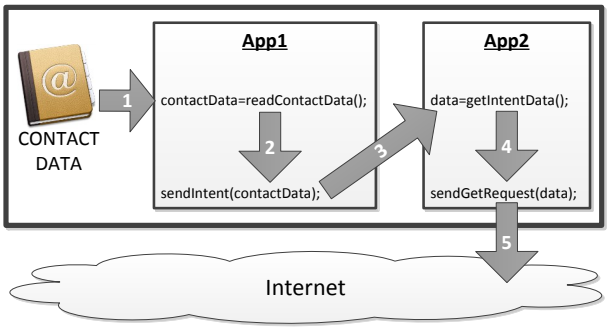

Fig. 3. Data Leak Collusion Attack. App 1 can access the contacts database, but can not connect to the internet. App 2 has access to the Internet, but not to the contact list. In this attack, App 1 reads contacts data (1) and sends it to App 2 via intent (3). App 2 is then allowed to send the data it received to a remote server (5). Note that in order to prevent this attack, one must be able to track the complete flow (1-5). DROIDFORCE does it in two phases: using static analysis to model dependencies between sources and sinks (1-2 and 4-5), and combining this information with runtime observations (3).

Figure 2 shows a simple communication diagram for this scenario. Three apps contain calls to `sendTextMessage()`. When the first application asks the decision point for permission, the PDP's counter is still zero, so it gets incremented and the request is granted. The second application's request is also allowed and the counter is increased once more. Since the example policy however only allows at most two SMS messages to be sent, the request made by the third app (assuming it happens within 24 hours from the first request) is rejected and no further SMS message is sent until 24 hours from the first request has passed.

The events sent to the decision point are also parameterized with runtime data. In the example, the decision point not only receives the fact that an SMS message is to be sent, but also the text message and the target telephone number. This allows the decision point to decide rich parameterized *expressive* policies. One could, for instance, only impose a limit on the number of messages sent to premium-rate numbers, but allow an arbitrary number of messages to normal-rate numbers. Another usage example would be to prevent target SMS spam by imposing a general limit on the number of messages per telephone number. Furthermore, one could also enforce restrictions in the carrier's plan for not exceeding the number of included SMS messages per month to avoid extra costs.

B. Example 2: Data Leakage Aggregation

In the second example of Section I, multiple applications send different pieces of data to the same server, thereby allowing the attacker to aggregate the various pieces of information, which is to be prevented. DROIDFORCE generates an event for the centralized decision point whenever an application sends a single piece of data. The decision point can keep track of this information, observe the total information leakage per remote server (or IP-address range) and disallow further transmissions if the accumulated leakage violates certain user-defined criteria (e.g. a threshold on the amount of data sent or a constraint on data aggregation). This allows for expressive policies that do not necessarily impact the functionality of single benign applications, but still guarantee a certain level of privacy for the overall system. A weather application, for instance, is required to disclose the user's location, but can do so anonymously. As long as this data cannot be aggregated with other information, the leak is not substantial in itself. This is an example of the *data-centric* policies that can be enforced by DROIDFORCE.

Of course, DROIDFORCE's ability to detect data aggregation is bounded by its ability to correlate remote targets, e.g., if the different data records are sent to the same DNS name and can easily be linked. With minimal effort, more sophisticated correlation techniques could be integrated into DROIDFORCE.

C. Example 3: Collusion Attack for Data Leakage

DROIDFORCE's data flow tracking capabilities go beyond simple intra-application intra-component flow analysis. Consider the third example in section I in which an attacker tries to steal a user's contact list using multiple colluding applications. Spreading the functionality among multiple apps avoids having a single application requesting access to, e.g., both the list of contacts and the Internet, which may look suspicious to a careful user who thoroughly reviews the permissions of an application before installing it. However, two seemingly innocent applications - one only requesting access to the contact database and one only to the Internet - can still collude to achieve the originally intended data leak. This attack cannot be prevented by the default Android permission system [15].

Figure 3 illustrates an example of such an attack. In app 1, the user's contact data is read, passed on through the application, and then sent to app 2 using an intent, a common mechanism for inter-app communication on the Android platform. App 2 receives the intent, reads the data contained in it (i.e. the contact database), and sends it to a remote server using an *http get* request. Analyzing each application in isolation would not be sufficient to detect this flow. More precisely, because an intent could possibly contain any kind of data, including contact data, a conservative static analysis of app 2 would prevent *any* invocation of an *http get* request after receiving the intent, regardless of the content. This means that legitimate invocations would be denied as well.

DROIDFORCE's approach, in contrast, combines the (statically computed) data flows of both applications at runtime. Firstly, when app 1 sends the intent, the enforcement framework attaches an additional parameter to it, called CONTACT_DATA and set its value to $true$. This happens because, according to previously computed static analysis results, data leaving

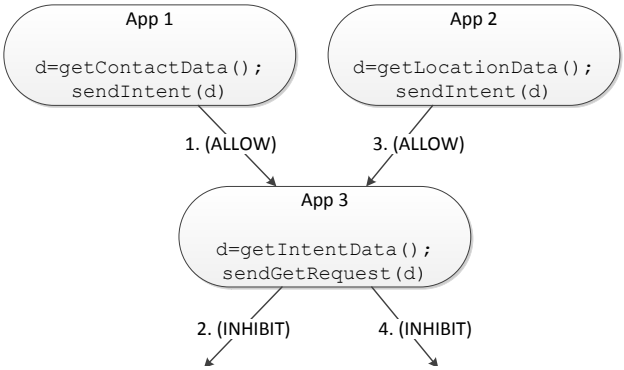

Fig. 4. Data-flows context-insensitivity example. The arrow labels indicate the order of execution and, in parenthesis, the PDP response for the policy "no contact data can be sent over the Internet". Notice that the second getRequest operation (number 4) is also denied, although legit.

app 1 at this particular endpoint (sink) may possibly depend on the result of the invocation of a certain source method (readContactData()) which returns contact data.

Secondly, when app2 receives the intent, the enforcement framework checks for the presence of special data parameters in the intent, like CONTACT_DATA. If such parameters are found (and set to $true$), then the table that stores app2's static analysis results is updated. More precisely, the source instruction currently being executed (or callback method receiving the intent) is now also associated with each source of the received data (CONTACT_DATA in the example). As a consequence, every sink instruction in app2 that, according to the same table, depends on this particular source, is now potentially leaking CONTACT_DATA. In other words, the table containing the statically computed data flows is dynamically extended with inter-app flows by building the transitive closure over the individual intra-app flows.

Finally, when app2 attempts to send some data via the Internet, DROIDFORCE uses app2's table to identify the origin of data being sent (CONTACT_DATA) and adds the relative parameter to the PDP request. Considering that the policy states that no contact data should be disclosed over the Internet, the PDP will deny such requests, preventing the leakage. This is a detailed example of how DROIDFORCE can prevent complex collusion attacks that involve inter-application flows of data.

D. Discussion

The described approach for tracking data flows between components inside and across application boundaries is fast since applications must only be analyzed once in isolation at instrumentation time. In contrast, considering all possible combinations of applications a user might have on his phone, as in existing approaches, is a substantial effort that may even be infeasible for a realistic number of applications [16].

DROIDFORCE's approach is context-insensitive. Once sensitive data is received by a certain source instruction, every execution of a sink that depends on that source is assumed to possibly leak that sensitive data, even if meanwhile the source is invoked again with non-sensitive data. Inside one application, the table of source-sink dependencies is computed statically, so every value ever arriving at a source potentially arrives at the connected sink(s).

For example, consider the *worst-case* scenario depicted in Figure 4 for the policy "contact data can not be sent over the Internet". In the example, two applications (app 1 and 2) send data to a third application which, in turn, sends all data it receives to the Internet. After receiving the contact data (1), app 3 tries to send it over the Internet (2), but the request is denied by the PDP. Afterward, app3 receives *through the same source instruction*, location information from a different app (3). Because the policy only forbids contact data from being leaked, the event should be allowed. DROIDFORCE, however, denies also this transmission (4). The reason is that our static tracking associates the outgoing http request with every data received via this particular callback (1 and 3). Since a context-insensitive analysis cannot keep track of which of two previously received pieces of data is being sent at the moment, it must conservatively assume the worst: the data being sent now (4) could be any of those received so far, including contact data (1). Therefore, DROIDFORCE needs to always block every further http request, even if it would only send the location data, which is allowed by the policy.

Note that this imprecision only happens if *both different* pieces of sensitive data have been received by the *same* component. Such behavior is rare in realistic applications since there is usually a separation of duty between components for architectural reasons. Moreover, the imprecision could be overcome by applying a fully dynamic data flow tracking, instead of relying on statically computed paths; this, on the other hand, would adversely affect runtime performance.

The complete policy decision is centralized in DROIDFORCE. While this requires the policy enforcement code to generate requests for all sensitive events even if they are not addressed by the currently enforced policy, this does not adversely affect performance in practice, as we show in Section V-A. On the other hand, it gives DROIDFORCE the flexibility to exchange the policy at any time without having to re-instrument the controlled applications with new enforcement / event generation code. Furthermore, it does not require any changes to the operating system's binder module, which transfers intents between applications; this makes DROIDFORCE usable without root access to the device or re-flashing and does not interfere with Android updates shipped to the phone.

IV. POLICY LANGUAGE

Considering the example requirements described in the previous sections, we decided to use OSL (Obligation Specification Language) [12] as policy specification language. OSL is a first-order temporal-logic language (e.g. "A until B"), with support for cardinality (e.g. "A at most 3 times") and time constraints (e.g. "A within 30 seconds"). Policies are written in form of Event-Condition-Action rules: if a system event **E** is detected and allowing its execution would make OSL condition **C** true, then action **A** should be performed. Action **A** states whether the execution of the event should be allowed or not, and if any additional action should be executed. Additional actions could for instance include asking the user for permission or reporting the violation via a popup window. By default any event is allowed, unless a mechanism requires its inhibition.

For instance, an appropriate policy for the first example of the previous section (premium SMS) could be "no more than

2 SMS messages per day may be sent to +01-234-5678". In our framework, this translates into

E: *sendTextMessage* to +01-234-5678
C: not ($replim$ (24 hours, 0, 1, *sendTextMessage* to +01-234-5678))
A: Inhibit

The semantics of the not operator is intuitive, whereas $replim(time, min, max, \phi)$ is true if ϕ has been true in the past $time$ at least min times and at most max times. Otherwise the operator evaluates to false. In this example, a message to +01-234-5678 can be sent if, in the previous 24 hours, other SMS to the same number have been sent at least 0 times and at most 1 time. Recall that in order for the trigger event to be allowed, the condition must evaluate to false, which explains the presence of the not operator. The concrete syntax of the policy is a bit more complex but reflects the same structure:

```
<preventiveMechanism name="limitSMS">
  <description>
    No more than n SMS to specific number per day
  </description>
  <trigger action="sendTextMessage"
           isTry="true" >
    <paramMatch
            name="destination"
            value="+01-234-5678" />
  </trigger>
  <condition>
    <not>
      <repLim amount="24" unit="HOURS"
              lowerLimit="0" upperLimit="1"
        <eventMatch
            action="sendTextMessage"
            isTry="false" >
            <paramMatch
              name="destination"
              value="+01-234-5678" />
        </eventMatch>
      </repLim>
    </not>
  </condition>
  <authorizationAction name="default" >
    <inhibit />
  </authorizationAction>
</preventiveMechanism>
```

Notice that our system distinguishes between *desired* and *actual* events. A desired event is an attempt of executing an event that has been intercepted by our PEP component. An actual event is an event that has been observed by our PEP but that has already happened or is going to be executed anyway and cannot be inhibited. In our concrete language this distinction is modeled by the flag $isTry$, which is true for desired events and false for actual events. The interpretation of the above policy is then "if the app is trying ($istry = true$) to send an SMS and in the last 24 hours more than 1 SMS has been actually ($istry = false$) sent, then deny the request". Given an application that tries to send SMS in a loop, this would mean that every 24 hours the app would succeed in sending two SMS. If on the other hand, we change the $istry$ flag in the condition to $true$, then also failed attempts would be counted and the app wouldn't be able to send more than 2 SMS in total, unless it stops trying for at least 24 hours. This behavior is not uncommon in premium-SMS trojans [14].

Notice that the conditions are expressed using the past variant of temporal logic operators. This is because ECA policies are enforced at runtime, where decisions can only be taken based on what already happened and not on possible future events. For this reason, specification and enforcement of liveness properties [17] are possible only in a limited time-bounded way, e.g. "the location data history must be deleted after at most a day".

As described before (cf. Section III-C), a message sent by the PEP to the PDP contains the name and all the parameters of the intercepted event, plus an additional one for each sensitive data in the system, like unique identification number (IMEI), location data, contact data, etc. Each of them is set to true if the target of the event possibly contains data of that kind. With this information, we can write a policy like "Advertisement server leadbolt.com should not receive both IMEI and GPS position" (second example), as

E: *network_send()*
C: target URL contains leadbolt.com and ((parameter IMEI_DATA = true and $eventually$ (httpRequest to "leadbolt.com" with parameter GPS_DATA=true)) or (parameter GPS_DATA = true and $eventually$ (httpRequest to "leadbolt.com" with parameter IMEI_DATA=true)))
A: Inhibit

where $eventually$ means "at least once so far" and $network\_send()$ is the high level event signaled by the PEP in correspondence of any API call that attempts to send data over a network connection, such as the constructor of *java.net.URL* for http get requests or writing on a stream retrieved from *java.net.Socket.getOutputStream()* for binary network data. The idea of this policy is that if we are trying to send one of the two pieces of sensitive information (location or unique identification) and the other one has been already sent in the past, then the event should be forbidden. The respective concrete policy can be found in Appendix.

Because the events received by the PDP could be generated by different applications, policies in DROIDFORCE control the use of sensitive data not only inside a single application, but also across application boundaries. This is particularly relevant for the third example, in which an attacker tries to steal user's contact list using one application to retrieve it and another one (possibly hijacked [9]) to send it over the Internet. In this scenario, the policy "every attempt of sending contacts data over the Internet must be manually approved by the user (via popup notification)" would be translated into

E: *network_send()*
C: (parameter CONTACT_DATA = true)
A: If (user_prompt(*"Allow this app to send contacts data?"*)==YES) then allow, otherwise inhibit.

In this section, we focused on the policies for the three running examples, but the expressiveness of the language allows for way more complex conditions [12], [18]. Though writing a policy in OSL requires a degree of expertise that typical users do not have, results from existing work can be used to refine informal requirements to concrete policies [19].

Package Name (Version Code)	# Instr. Orig.	# Instr. Orig. Soot	# Instr. Instrum. Soot	Overhead Instr. (perc.)	Static Analysis [sec.]	Instrum. [sec.]
bg.angelov.send.my.location (9)	107,938	112,819	113,814	995 (0.9%)	7.6 (+/-3.6)	23.9 (+/-1.6)
buy.more.chuck (3)	16,313	17,256	19,699	2443 (14.2%)	22.7 (+/-0.9)	2.5 (+/-0.2)
com.advancedprocessmanager (59)	70,216	86,823	90,251	3,428 (3.9%)	48.5 (+/-4.0)	8.4 (+/-0.7)
com.bfs.papertoss (7005)	93,562	106,520	109,360	2,840 (2.7%)	28.1 (+/-0.9)	14.0 (+/-0.8)
com.mymobileprotection20 (19)	56,958	65,739	70,553	4,814 (7.3%)	19.0 (+/-8.5)	4.7 (+/-0.7)
com.xmm.surgery (11)	84,252	102,843	105,626	2,783 (2.7%)	31.3 (+/-2.0)	11.2 (+/-0.3)
org.me.SendSMS (8)	34,160	38,948	41,584	2,636 (6.8%)	13.4 (+/-8.5)	2.1 (+/-0.7)
org.nastysage.blacklist (2016177433)	89,417	92,633	93,872	1,237 (1.4%)	18.2 (+/-7.5)	6.3 (+/-1.2)
tv.twitch.android.viewer (11)	29,077	33,038	33,887	849 (2.6%)	18.2 (+/-0.5)	9.3 (+/-1.2)
www.eidolonstudio.puzzleBall (8)	22,096	26,767	28,204	1,437 (5.4%)	24.9 (+/-0.7)	5 (+/-0.4)
com.flash.light.flashlight (8)	194801	207215	208941	1726 (0.8%)	13.1 (+/-7.0)	31.3 (+/-2.1)
ro.robisoft.android.flashlight (1)	182	183	998	815 (445.4%)	1.2 (+/-1.3)	2.0 (+/-1.7)

TABLE I. OVERHEAD INTRODUCED BY OUR FRAMEWORK IN TERMS OF ADDITIONAL INSTRUCTIONS AND TIME REQUIRED FOR STATIC ANALYSIS (FLOWDROID) AND FOR INSTRUMENTATION (SOOT).

V. EVALUATION

In this section, we evaluate DROIDFORCE. Section V-A deals with DROIDFORCE's performance, both for instrumenting new applications before they are deployed on the phone and for running the resulting apps on the device, whereas in Section V-B we discuss the security of DROIDFORCE with respect to the Android environment and some limitations of our current implementation.

A. Performance

Our performance evaluation considers the following research questions:

- RQ1: How long does the instrumentation phase take?
- RQ2: How many additional instructions are injected in an application during the instrumentation phase?
- RQ3: How much slower is the instrumented application w.r.t. the original version?

We tested our approach on randomly-picked real-world Android applications taken from the official Google Play Store. In order to answer the first two research questions, we compared the size of the applications in terms of bytecode instructions before and after the instrumentation, as shown in Table I. The first column contains the package name of the application, together with its version code, as defined in the manifest file. The second column shows the number of instructions in the off-the-shelf application, before applying any kind of transformation to it. The process of converting the application code into the intermediate language (jimple) used by the Soot instrumentation tool and then back into Android bytecode introduces some additional instructions, that have nothing to do with our DROIDFORCE framework. For this reason, in our evaluation we factor them out by comparing the number of instructions after running our DROIDFORCE instrumentation (fourth column) with the number of instructions after performing a conversion dex→jimple→dex without any modification (third column). Notice that the application after this conversion, although bigger in size, is semantically equivalent to the original one.

Based on these values, we computed the overhead induced by our infrastructure in terms of additional instructions (fifth column). As expected, this proved to be reasonably small, confirming that DROIDFORCE does not introduce undue overhead on an application.

In order to answer RQ1, we also measured the time required to instrument the applications. All timings were averaged over ten runs. The values in brackets in the table show the standard deviation. The tests were carried out on a MacBook Pro computer running MacOS X version 10.7.4 on a 2.5 GHz Intel Core i5 processor and 8 GB of memory. Oracle' s JDK implementation was used in its default configuration, except for the maximum heap size which was increased to 6 GB.

Furthermore, Table I shows the average time required to perform the static data flow analysis (sixth column) and to instrument the policy enforcement code into the application (seventh column). The values in brackets indicate the standard deviation. The complete process of analyzing and instrumenting never takes more than one minute with an average of 34.4 seconds, which confirms that even an on-the-fly instrumentation on an external server whenever a user installs a new application on his phone would be feasible.

In terms of RQ2, we start from a constant base overhead of 815 instruction that DROIDFORCE always injects into an application. These instructions are communication interfaces between the enforcement code and the decision point application. In addition to them, for each sensitive instruction in the application, we inject some "wrapping" code, that includes the creation of the PDP request, the lookup table for source-sink dependencies and the enforcement of the PDP decision. For the sake of our proof-of-concept experiment we used the most frequently used source and sink API methods in malware apps from the literature [20]. These APIs also contain the APIs required for specifying the three example policies described in Section III. Additionally, DROIDFORCE also injects fake initialization for variables that would have been written in the skipped code, in order to increase application stability and avoid crashes where possible.

On average, for the randomly-picked applications we measured an overhead in terms of additional instructions of less than 4%. This number, however, needs to be interpreted with a grain of salt: the size of the application (for the constant overhead) and the number of sensitive to-be-instrumented instructions significantly affect the individual results.

In the lower part of Table I we show the same analysis for two flashlight applications manually picked from the Google Play Store. Although they both offer roughly the same functionality, one (*com.flash.light.flashlight*) uses a large advertisement library known to leak sensitive information and one (*ro.robisoft.android.flashlight*) does not contain any advertisement . Since the latter does not contain any sensitive operations and is very small in size (183 instructions), the

baseline is exceptionally low, which leads to a total overhead after instrumentation of about 450% with no single instrumented operation. Note that for this particular example, one could use the ad-hoc solution of not adding the interface code if there is no sensitive operation in the target application. Still, when at least one of such calls exists, all 815 instructions must be added anyway.

Nevertheless, the real world applications that we randomly picked form the market were in general big enough to mitigate the impact of the constant overhead. The "variable" overhead, in contrast, is highly dependent on the number of sensitive instructions in the application, which is usually related to the advertisement libraries used in it and to the semantics of the application (e.g. a social network app should in general contain more sources and sinks than a flashlight app).

In terms of RQ3, we tested every instrumented application on a Nexus 4 phone (Android version 4.4.2), and none of them exhibited any perceivable slowdown in the user experience. Moreover, although possible, during the tests none of the applications crashed, partially thanks to the additional initializations injected by DROIDFORCE. Based on these preliminary experiments we have good reasons to believe that our approach is indeed usable on real-world applications.

B. Security analysis

DROIDFORCE works by instrumenting API calls at the bytecode level. Methods executed in native libraries are not monitored and could, in principle, evade our analysis. A simple solution would be to prevent the execution of any native code at all by instrumenting the bytecode accordingly. This would obviously compromise the functionality of the application, although it is known to the literature that only about 4.52% of all application in the Google market use native code at all [21]. Alternatively, native calls could be over-approximated or modeled using external domain knowledge. For the most common native methods such as `System.arraycopy`, DROIDFORCE already provides such models. Lastly, solutions for instrumenting native libraries can be found in the literature [22], but various challenges such as self-modifying code or precisely modeling complex pointer computations remain.

For a complete enforcement, DROIDFORCE must make sure to monitor all sensitive API calls possible in the Android framework. However, this is a non-trivial challenge since there exist around 110 000 public API methods in Android 4.2. Recent work such as SuSi [20] is able to provide comprehensive lists of data sources and sinks through the use of machine-learning. SuSi's classification has both a recall and precision of over 90% which means that over 90% of all sources and sinks are actually flagged as such. All the API calls in this list that offer the same functionality are grouped in categories, like UNIQUE_IDENTIFIER for sources or SMS_MMS for sink APIs. This is useful to formulate policies in terms of abstract events like "send via Internet" or "send SMS" instead of extensively listing all the possible technical alternatives. In addition, relying on the SuSi list also prevents attackers from easily circumventing the restrictions imposed by DROIDFORCE by simply using alternative methods/APIs for the same functionality that are not covered by our tool.

In this work we do not consider the attacker model of a user wanting to circumvent usage control restrictions on her own phone. We instead assume the attacker to be a malicious application (or set of applications) that tries to steal information or, in general, act against user's will, specified in form of policies. For this reason, it is safe for our analysis to assume that our PDP application (cf. Section III) is installed and thus protected by the application isolation built into the Android framework. Other applications have no access to the PDP's memory region or private files and thus cannot tamper with the policies registered in the PDP. Additionally, as stated before, we assume that every application in the system has been previously instrumented using our approach. Lastly, we assume that no malicious application is running with root permissions, otherwise, like in any system, no concrete security claim could be made at all. This also implies that malicious application cannot uninstall or replace the decision point application.

Since the decision point does not provide any publicly accessible APIs for creating, modifying, or deleting policies, malicious applications cannot disable restrictions the user has placed upon them. The policies themselves are provided as files and are thus only accessible to the PDP application itself. All administration interfaces for modifying policies are only accessible to applications signed with the same key as the PDP, i.e., applications provided by a trusted manufacturer. This signature restriction is configured in the PDP's manifest file and enforced by the Android operating system.

An application could, however, execute a large number of protected operations and thereby generate many requests for the decision point for launching a denial-of-service attack. This can be easily overcome either by adding a small but increasingly bigger delay in the application after denying the execution of an event or by writing a simple policy that kills the application if it generates more than a certain maximum amount of events in a limited time.

A malicious application could also try to provide a fake implementation of the decision point that simply allows all protected operations regardless of any policy. DROIDFORCE protects against such attacks by sending policy events using *explicit* intents. With this technique, the operating system's intent dispatcher is forced to a single receiver class, namely DROIDFORCE's decision point implementation.

DROIDFORCE assumes that all code being executed is available at analysis time. If a malicious application downloads additional code from the Internet and executes it, leaks in this code are not detected. To solve this problem, we could detect such code loading and re-route the traffic through a server where we analyze and instrument the loaded code (i.e. run the static part of DROIDFORCE) on-the-fly. Given that analyzing and instrumenting the code takes less than a minute as shown in Section V-A, this is reasonable as an on-the-fly-approach. We plan to do this as part of our future work. The instrumentation part of DROIDFORCE could potentially also be done on the phone, but the static analysis usually requires more hardware resources than what is currently available on mobile devices.

Since by design our analysis only detects policy violations due to executed events, DROIDFORCE is not able to detect data leaks due to events *not* being executed. Assume a malicious application contains the code from Listing 1. In this case,

```
1  void attack() {
2    if (secret == 42)
3      sendTextMessage(...);
4  }
```

Listing 1. Field Library Interface

the attacker can infer that the secret was not 42 if he does not receive the SMS message. Such cases are currently not covered by our approach. Instead, one can detect and react to the violation of a timed obligation, e.g., "A must happen at most ten minutes after B". Anyway, we have not discovered any such attacks in real-world applications.

Lastly, for data-flow policies, DROIDFORCE inherits the limitations of the FlowDroid data flow tracker on which it is built. In FlowDroid, reflective method calls can be analyzed if the method to be invoked and the name of the class containing this method are constant.

VI. RELATED WORK

Android applications can be analyzed for policy violations in a purely static way. Data flow trackers such as FlowDroid [5], Scandal [8], LeakMiner [7], or AndroidLeaks [23] detect intra-app data flows without the need for executing the application. They are however prone to false positives and can only over-approximate inter-app data flows by flagging all data leaving the current application as a potential leak. Approaches such as CHEX [6], WoodPacker [24] or ComDroid [25] scan applications for potential confused-deputy attacks where an app is able to access sensitive data in the context of a vulnerable app. These approaches are only intra-application, intra-component and sometimes even intra-procedural, where DROIDFORCE is able to enforce complex data-centric policies between components in an app and even app comprehensive. Epicc [26] is a tool for statically detecting inter-component and inter-application data flows on the Android platform. DROIDFORCE is more precise since it only reports flows that actually occur and lead to policy violations while Epicc reports all statically possible flows even if the respective code paths leading to such flows are never taken at runtime.

AppGuard [27] allows users to enforce access-control policies on Android applications by dynamically revoking permissions from apps after they are installed. On stock Android systems, app can only be installed with all the permissions they require or not at all. AppGuard overcomes this limitation, but is not capable of enforcing data-centric policies such as "allow this application to access the location data and the Internet, but prevent it from sending out the location data to a remote adversary". Kynoid [28] is a dynamic data flow analysis tool for Android based on TaintDroid [29]. It works by replacing the Dalvik virtual machine with an implementation that tracks taint bits along with register values. This requires changes to the underlying operating system, in contrast to DROIDFORCE, which works on unmodified stock Android phones. Feth et al. [30] proposed a usage control solution for Android similar to ours. Their work also leverages TaintDroid, allowing them to track system-wide data-flows for every installed application, but consequently sharing TaintDroid's limitation of requiring a dedicated modified operating system.

XManDroid [31] was designed to prevent privilege escalation and collusion attacks by enforcing policies on the communications between applications, e.g., banning an application that has access to the user's location from interacting with an application that is allowed to access the Internet. In contract to DROIDFORCE, such policies are very coarse-grained and not data-centric.

Aurasium [32] repackages applications and introduces an intermediate layer between the framework's native code libraries and the operating system kernel inside the application process. Just like DROIDFORCE, this does not require any changes to the operating system. While Aurasium can also detect protected operations in native code, it is highly dependent on undocumented internal interfaces between the Android libraries and the kernel which may change in future Android versions without notice. DROIDFORCE does not need such assumptions.

AppSealer [33] combines static- and dynamic-code analysis techniques to patch the apps's bytecode in order to mitigate component hijacking attacks. Like DROIDFORCE, they rely on Soot, but their focus is only on component hijacking vulnerabilities. DROIDFORCE is able to enforce complex data-centric policies, including timing and/or sequence based policies, which is required for practical system-wide enforcement.

Solutions for enforcing security policies based on inline reference monitoring have also been proposed for other (mobile) environments such as J2ME or Windows Mobile [34], [35]. To the best of our knowledge, none of these approaches combine IRM for usage control with data flow tracking that enforces complex policies on multiple applications at the same time *without* requiring root access to the phone.

VII. CONCLUSIONS

We presented DROIDFORCE, an approach for enforcing complex, data-centric, system-wide policies on Android applications. Unlike previous approaches, DROIDFORCE can prevent collusion and data aggregation attacks by keeping a global policy state. Furthermore, rich data-centric policies such as banning SMS messages depending on the telephone number can be expressed. No modifications to the Android operating system are required, nor rooting the phone. The PEP code is injected into applications in less than a minute and introduces no perceivable delay at runtime. We thus consider DROIDFORCE an important step towards securing smartphone platforms.

As future work, we plan to extend DROIDFORCE to automatically re-instrument applications when they are updated. Updates replace the app and thus destroy the injected enforcement code. This can be solved by relaying all app installations and updates through a trusted instrumentation server on the Internet. Instrumenting applications directly on the phone is also an interesting direction of future research which is challenging due to the resource-constrained nature of the platform.

Acknowledgements: Thanks to Cornelius Moucha for the technical support. This work was supported by a Google Faculty Research Award, by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED and by the DFG within the projects RUNSECURE and p-SADAN (PR-1266/1-2), which are associated with the DFG Priority Programme 1496 "Reliably Secure Software Systems – RS^3".

REFERENCES

[1] International Data Corporation, "Worldwide quarterly mobile phone tracker 3q12," Nov. 2012, http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37.

[2] I. Security, "Mcafee mobile security report," 2012, http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf.

[3] E. Chien, "Motivations of recent android malware," Feb. 2014, http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf.

[4] "Top 10 spy software," April 2014, http://www.top10spysoftware.com/.

[5] 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 *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, p. 29.

[6] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *CCS 2012*.

[7] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*, 2012, pp. 101–104.

[8] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST 2012: Mobile Security Technologies 2012*. IEEE, May 2012.

[9] T. Cannon, "No-permission android app gives remote shell," https://viaforensics.com/security/nopermission-android-app-remote-shell.html, 2011.

[10] K. Makan, "Path traversal vulnerability in oi file manager for android," http://blog.k3170makan.com/2014/02/path-disclosure-vulnerability-in-io.html, 2014.

[11] A. Alkassar, S. Schulz, C. Stble, and S. Wohlgemuth, "Securing smartphone compartments: Approaches and solutions," in *ISSE 2012*. Springer Fachmedien Wiesbaden, 2012, pp. 260–268.

[12] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter, "A policy language for distributed usage control," in *Proc. 12th Europ. Conf. on Research in Computer Security*. Springer-Verlag, pp. 531–546.

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proc. 1999 Conf. of Centre for Advanced Studies on Collaborative Research*. IBM Press.

[14] M. Spreitzenbarth, "Detailed analysis of android.fakeregsms.b," http://forensics.spreitzenbarth.de/2012/02/03/detailed-analysis-of-android-fakeregsms-b/, 2012.

[15] C. Marforio, A. Francillon, and S. Čapkun, "Application collusion attack on the permission-based security model and its implications for modern smartphone systems," ETH Zurich, Tech. Rep. 724, April 2011.

[16] D. Hausknecht, "Variability-aware Data-flow Analysis for Smartphone Applications," Master's thesis, University of Passau, 2013.

[17] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, Mar. 1977.

[18] A. Pretschner, E. Lovat, and M. Büchler, "Representation-independent data usage control," in *Proc DPM 2011*, 2011, pp. 122–140.

[19] P. Kumari and A. Pretschner, "Deriving implementation-level policies for usage control enforcement," in *Proc. CODASPY '12*.

[20] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. NDSS 2014*, Feb. 2014.

[21] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. NDSS 2012*, Feb. 2012.

[22] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 261–270.

[23] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *Proc. TRUST 2012*, 2012.

[24] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proc. NDSS 2012*.

[25] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proc. MobiSys 2011*, 2011.

[26] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proc. USENIX 2013*, 2013.

[27] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard - real-time policy enforcement for third-party applications," Tech. Rep., 2012.

[28] D. Schreckling, J. Posegga, J. Kstler, and M. Schaff, "Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android," in *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*, 2012.

[29] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI*, 2010, pp. 393–407.

[30] D. Feth and A. Pretschner, "Flexible data-driven security for android," in *Proc. IEEE Intl. Conf. Software Security and Reliability SERE'12*.

[31] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," Technische Universität Darmstadt, Tech. Rep., Apr. 2011.

[32] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: practical policy enforcement for android applications," in *USENIX Security 2012*.

[33] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," in *Proc. NDSS 2014*.

[34] A. Castrucci, F. Martinelli, P. Mori, and F. Roperti, "Enhancing java me security support with resource usage monitoring," in *ICICS*, 2008, pp. 256–266.

[35] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe, "A flexible security architecture to support third-party applications on mobile devices," in *Proc. 2007 ACM Workshop on Computer Security Architecture*, ser. CSAW '07.

APPENDIX

```
<preventiveMechanism name="noImeiAndGPStoAds" >
  <trigger action="httpRequest">
    <paramMatch name="targetDomain"
      value="leadbolt.com"/>
  </trigger>
  <condition>
    <or><and>
      <xPathEval>
//event/parameter[@name='IMEI_DATA']/@value = 'true'
      </xPathEval>
      <not><always><not>
        <eventMatch action="httpRequest">
          <paramMatch name="targetDomain"
            value="leadbolt.com" />
          <paramMatch name="GPS\_DATA"
            value="true" />
        </eventMatch>
      </not></always></not>
    </and><and>
      <xPathEval>
//event/parameter[@name='GPS_DATA']/@value = 'true'
      </xPathEval>
      <not><always><not>
        <eventMatch action="httpRequest">
        <paramMatch name="targetDomain"
          value="leadbolt.com" />
        <paramMatch name="IMEI\_DATA"
          value="true" />
        </eventMatch>
      </not></always></not>
    </and></or>
  </condition>
  <authorizationAction name="default" >
    <inhibit />
  </authorizationAction>
</preventiveMechanism>
```