Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Automated Dynamic Enforcement of Synthesized Security Policies in Android

Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand and Sam Malek
{hbagheri, asadeghi, rjabbarv, smalek}@gmu.edu

Technical Report GMU-CS-TR-2015-5

Abstract

As the dominant mobile computing platform, Android has become a prime target for cyber-security attacks. Many of these attacks are manifested at the application level, and through the exploitation of vulnerabilities in apps downloaded from the popular app stores. Increasingly, sophisticated attacks exploit the vulnerabilities in multiple installed apps, making it extremely difficult to foresee such attacks, as neither the app developers nor the store operators know a priori which apps will be installed together. This paper presents an approach that allows the end-users to safeguard a given bundle of apps installed on their device from such attacks. The approach, realized in a tool, called DROIDGUARD, combines static code analysis with lightweight formal methods to automatically infer security-relevant properties from a bundle of apps. It then uses a constraint solver to synthesize possible security exploits, from which fine-grained security policies are derived and automatically enforced to protect a given device. In our experiments with over 4,000 Android apps, DROIDGUARD has proven to be highly effective at detecting previously unknown vulnerabilities as well as preventing their exploitation.

1 Introduction

The ubiquity of smartphones and our growing reliance on mobile apps are leaving us more vulnerable to cyber security attacks than ever before. According to the Symantec's Norton report [20], in 2013 the annual financial loss due to cybercrime exceeded $113 billion globally, with every second 12 people become the victim of cybercrime. An equally ominous report from Gartner [35] predicts 10 percent yearly growth in cybercrime-related financial loss through 2016. This growth is attributed in part to the new security threats targeted at emerging platforms, such as Google Android and Apple iPhone, as 38% of mobile users have experienced cybercrime [20]. This is, though, nowhere more evident than in the Android market, where many cases of apps infected with malwares and spywares have been reported [48].

In this context, smartphone platforms, and in particular Android, have emerged as a topic *du jour* for security research. These research efforts have investigated weaknesses from various perspectives, including detection of information leaks [23, 29, 33, 41], analysis of the least-privilege principle [24, 25], and enhancements to Android protection mechanisms [16, 22, 27]. Above and beyond such security techniques that are substantially intended to detect vulnerabilities in a single application, researchers have recently investigated techniques tackling security vulnerabilities that arise due to the interaction of multiple applications, such as inter-component data leaks [37,38,55] and permission leaks [14,32], shown to be quite common in the apps on the markets.

While the prior techniques mainly aim to find security weaknesses in existing combination of apps, we are also interested in the dual of this problem, that is *what security attacks are possible given a set of vulnerable apps?* Many Android malwares are embedded in supposedly normal apps that aim to leverage vulnerabilities in either the platform or other apps on the market for nefarious purposes. If we could automatically generate security exploits for a given combination of apps, it would allow us to identify possible security attacks before the adversary, and thus protect our systems prior to the realization of such attacks.

In this paper, we propose a proactive scheme to develop Android security policies for vulnerabilities that occur due to the interaction of apps comprising a system. Our approach aims to automatically find vulnerabilities in a given bundle of apps and generate specifications of possible exploits for them, which then can proactively be applied as preventive measures to guard against yet unknown malicious behavior.

Specifically, we have developed an automated system called DROIDGUARD that combines scalable static analysis with lightweight formal methods. DROIDGUARD leverages static analysis to automatically infer security-relevant facts about software systems.[1] The app specifications are sufficiently abstract—extracted at the architectural level—to be amenable to formal analysis, and to ensure the technique remains scalable to real-world Android apps, yet represent the true behavior of the implemented software, as they are automatically extracted from the app bytecode, and appear sufficiently detailed to express subtle inter-app vulnerabilities.

DROIDGUARD then uses a SAT-based engine to analyze the system model against compositional security properties and generate potential attack scenarios. In fact, it mimics the adversary by leveraging recent advancements in constraint solving techniques to synthesize possible security exploits, from which fine-grained security policies are then derived and enforced for each particular system. The synthesis of system-specific security policies allows the user to proactively deploy preventive measures prior to the discovery of those exploits by the adversaries.

To summarize, this paper makes the following contributions:

- *Formal Synthesis of Security Policies:* We introduce a novel approach to synthesize specifications of possible exploits for a given combination of apps, from which system-specific security policies are derived. The policy synthesizer relies on a fully analyzable formal model of Android framework and a scalable static analysis technique extracting formal specifications of Android apps.

- *Runtime Enforcement of Security Policies:* We develop a new technology to automatically apply and dynamically enforce the synthesized, fine-grained policies (at the level of event messaging), specifically generated for a particular collection of apps installed on the end-user device.

- *Implementation of DROIDGUARD framework*: We describe DROIDGUARD—the first end-to-end system for fully automatic generation and enforcement of fine-grain, formally-precise, and system-specific security policies for inter-component vulnerabilities of real-world Android apps. DROIDGUARD is publicly available for download [4].

- *Experiments*: We present results from experiments run on 4,000 real-world apps as well as DroidBench2.0 test suite [3], corroborating DROIDGUARD's ability in (1) effective compositional analysis of Android inter-application vulnerabilities and generation of preventive security policies, that many of those vulnerabilities cannot be even detected by state-of-the-art security analysis frameworks; (2) outperforming other compositional analysis tools also in terms of scalability; and (3) finding multiple crucial security problems in the apps on the markets that were never reported before.

The remainder of paper is organized as follows. Section 2 provides an overview of Android. Section 3 motivates our research through an illustrative example. Section 4 provides an overview of DROIDGUARD. Sections 5, 6 and 7 describe the details of static model extraction, formal synthesis and dynamic enforcement of policies, respectively. Sections 8 and 9 present implementation and evaluation of the research. The paper concludes with an outline of the related research and future work.

2 Android Overview

This section provides a brief overview of the Android framework to help the reader follow the discussions that ensue.

The Android framework includes a full Linux OS based on the ARM processor, system libraries, middleware, and a suite of pre-installed applications. Android applications (apps) are written in Java and compiled into Dalvik bytecode [2]. Apps are then packaged as APK[2] files, used for distributing and installing them. Each app is executed in a separate instance of Dalvik Virtual Machine (DVM). [3]

Each Android APK includes a mandatory configuration file, called *manifest*. It specifies, among other things, the principal components that constitute the application, including their types and capabilities, as well as required and enforce permissions. The manifest file values are bound to the application at compile time, and cannot be changed afterwards, unless the application is recompiled.

Components are basic logical building blocks of apps. Each component can be invoked individually, either by its embodying application or by the system, upon permitted requests from other applications. Android defines four types of components: (1) *Activity* components provide the basis of the Android user interface. Each Application may have multiple Activities representing different screens of the application to the user. (2) *Service* components provide background processing capabilities, and do not provide any user interface. Play-

[1]By a software system, we mean a set of independently developed apps jointly deployed on top of a common computing platform, e.g. Android framework, that interact with each other, and collectively result in a number of software solutions or services.

[2]Android application package.

[3]In Android L, Dalvik has been substituted with the previously experimental Android runtime (ART) as a default environment.

ing a music and downloading a file while a user interacts with another application are examples of operations that may run as a Service. (3) *Broadcast receiver* components respond asynchronously to system-wide message broadcasts. A receiver component typically acts as a gateway to other components, and passes on messages to Activities or Services to handle them. (4) *Content Provider* components provide database capabilities to other components. Such databases can be used for both intra-application data persistence as well as sharing data across applications.

Inter-component communication (ICC) in Android is mainly conducted by means of *Intent* messages. An Intent message is an event for an action to be performed along with the data that supports that action. Component capabilities are then specified as a set of *Intent-Filters* that represent the kinds of requests handled by a given component. Component invocations come in different flavors, e.g., explicit or implicit, intra- or inter-applications, etc. Android's ICC allows for late run-time binding between components in the same or different applications, where the calls are not explicit in the code, rather made possible through event messaging, a key property of event-driven systems.

Permissions are the cornerstone for the Android security model. The permissions stated in the app manifest enable secure access to sensitive resources as well as cross-application interactions. When a user installs an app, the Android system prompts the user for consent to requested permissions prior to installation. Should the user refuse granting the requested permissions to an app, the app installation is canceled. No dynamic mechanism is provided by Android for granting permissions after app installation. Besides required permissions, the app manifest may also include enforced permissions that other apps must have in order to interact with this app. Android platform provides over 145 pre-defined permissions, and applications can also define their own permissions. Each permission is specified by a unique label, typically indicating the protected action. For instance, the permission label of `android.permission.SET_WALLPAPER` is required for an application to change the wallpaper.

The Android access control model is at the granularity of individual apps, and there is no mechanism to check security posture of the entire system. Such a permission mechanism has proved insufficient to prevent compositional security violations [16, 21, 25–27], since permissions may be misused, intentionally or unintentionally, as illustrated in the next section.

```
1  public class FindLocationActivity extends Activity {
2   public void onCreate (Bundle savedInstanceState ) {
3    Intent intent = new Intent();
4    intent.setAction("showLoc");
5    LocationManager lm = getSystemService(Context.
         LOCATION_SERVICE);
6    Location lastKnownLocation =
7    lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
8    intent.putExtra("locationInfo", lastKnownLocation.
         toString());
9    startActivity(intent);
10 }
```

Listing 1: Vulnerable app (App1): sends the retrieved location data to another component of the same app via implicit Intent messaging.

```
1  public class TelephonyActivity extends Activity  {
2   public void onCreate(Bundle savedInstanceState)  {
3    Intent intent = getIntent();
4    String number = intent.getStringExtra("PHONE_NUM");
5    String message = intent.getStringExtra("TEXT_MSG");
6    //if (hasPermission())
7    sendTextMessage(number, message);
8    ...}
9   void sendTextMessage (String num, String msg) {
10   SmsManager mngr = SmsManager.getDefault();
11   mngr.sendTextMessage(num,null,msg,null,null);
12  }
13  boolean hasPermission () {
14   if(checkCallingPermission("android.permission.SEND_SMS"
        )==PackageManager.PERMISSION_GRANTED)
15       return true;
16   return false;
17  }
18 }
```

Listing 2: Vulnerable app (App2): receives an Intent and sends a text message.

3 Motivating Example

To motivate the research and illustrate our approach, we provide an example of a vulnerability pattern having to do with ICC among Android apps. Figure 1 partially shows a bundle of two benign, yet vulnerable apps, installed together on a device.

App1 is a navigation application that obtains the device location (GPS data) in one of its components and then sends it to another component of the app via Intra-app Intent messaging. The Intent involving the location data (Listing 1, lines 3–8), instead of explicitly specifying the receiver component, implicitly specifies it through declaring a certain `action` to be performed in that component. This represents a common practice among the developers, yet an anti-pattern that may lead to unauthorized Intent receipt [18], as any component, even if it belongs to a different app, that matches the *action* could receive an implicit Intent sent this way.

On the other hand, the second app's vulnerability occurs on line 11 of Listing 2, where *TelephonyActivity* uses system-level API `SmsManager`, resulting in a message sent to the phone number previously retrieved from the Intent. This is a reserved Android API that requires special access permissions to the system's telephony service. Although TelephonyActivity has that permission, it also needs to ensure that the sender of the original Intent message has the required permission to use the SMS service. An example of such a check is shown in `hasPermission` method of Listing 2, but in this particular example it does not get called (line 6 is commented) to illustrate the vulnerability.

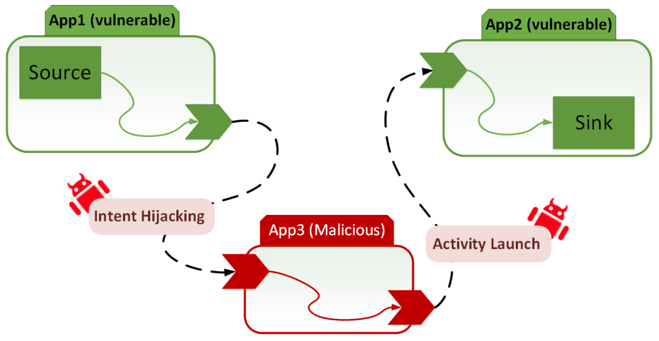

Figure 1: A potential malicious application—its signature automatically generated by DROIDGUARD—leverages vulnerabilities in other already installed benign applications to perform actions (like sending device location through text messages) that are beyond its individual privileges. As the Android access control model is per app, it cannot check security posture of the entire system. DROIDGUARD generates and enforces compositional policies that prevent such an exploit.

Given these vulnerabilities, a malicious app can send the device location data to the desirable phone number via text message, without the need for any permission. As shown in Figure 1, the malicious app, first hijacks the Intents containing the device location info from the first app. Then, it sends a fake Intent to the second app, containing the GPS data and adversary phone number as the payload. While the example of Figure 1 shows exploitation of vulnerabilities in components from two apps, in general, a similar attack may occur by exploiting the vulnerabilities in components of either single app or multiple apps.

Android's access control mechanism is at the level of individual apps, which is not sufficient to prevent compositional security violations, such as the one in this example. Moreover, since the malicious app does not require any security sensitive permission, it is easily concealed as a benign app that only sends and receives Intents. This makes the detection of such malicious apps a challenging task for individual security inspectors or anti-virus tools.

The above example points to one of the most challenging issues in Android security, i.e., detection and enforcement of compositional security policies to prevent such possible exploits. What is required is a system-level analysis capability that (1) identifies the vulnerabilities and capabilities in individual apps, and (2) determines how those individual vulnerabilities and capabilities could affect one another when the corresponding apps are installed together. In the next sections, we first provide an overview of DROIDGUARD and then delve into more details about its approach to address these issues.

4 Approach Overview

This section overviews our approach to automatically synthesize and enforce system-specific security policies for such vulnerabilities that occur due to the interaction of apps comprising a system. As depicted in Figure 2, DROIDGUARD consists of three main components: (1) The *Android model extractor (AME)* that uses static analysis techniques to automatically elicit formal specifications of the apps comprising a system; (2) The *analysis and synthesis engine (ASE)* that is intended to use lightweight formal analysis techniques [36] to find vulnerabilities in the extracted app models, and to generate specifications of possible exploits, and in turn, policies for preventing their manifestation; (3) The *Android policy enforcer (APE)* that enforces automatically generated, system-wide policies on Android applications.

The AME component takes as input a set of Android application package archives, called APK files. APKs are Java bytecode packages used to distribute and install Android applications. To generate the app specifications, AME first examines the application manifest file to determine its architectural information. It then utilizes different static analysis techniques, i.e., control flow and data flow analyses, to extract other essential information from the application bytecode into an analyzable specification language.

The ASE component, in addition to extracted app specifications, relies on two other kinds of specifications: a formal foundation of the application framework and the axiomatized inter-app vulnerability signatures. The Android framework specification represents the foundation of Android apps. Our formalization of these concepts includes a set of rules to lay this foundation (e.g., application, component, messages, etc.), how they behave, and how they interact with each other. It can be considered as an abstract, yet precise, specification of how the framework behaves. We regard vulnerability signatures as predicates that model Android inter-app vulnerabilities in relational logic, representing their essential characteristics as exhibited when the vulnerability is exploited. All the specifications are uniformly captured in the Alloy language [36]. Alloy is a formal specification language based on first-order relational logic, amenable to fully automated yet bounded analysis.

DROIDGUARD is designed as a plugin-based software that provides extension points for analyzing apps against different types of vulnerabilities. In order to analyze each app, we distill each known inter-app vulnerability into a corresponding formally-specified signature to capture its essential characteristics, as manifested when the vulnerability is exploited. Our current DROIDGUARD prototype supports inter-component vulnerabilities, such as Activity/Service launch, Intent hijack, privilege escalation, and information leakage [18, 21, 30]. Its plugin-based architecture supports the necessary extensions that can be provided by users

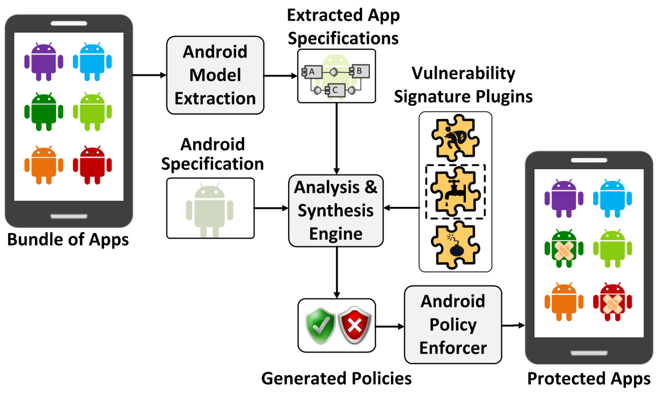

Figure 2: Approach Overview.

at anytime to enrich the environment.

Given these specifications, the ASE component analyzes them as a whole for instances of vulnerabilities in the extracted app specifications, and using formally-precise scenario-generating tools, such as Alloy Analyzer and Aluminum [36,42], it attempts to generate possible security exploit scenarios for a given combination of apps. Specifically, we go beyond the detection of vulnerabilities by asking: *what security attacks are possible given a set of vulnerable apps?*

Having computed system-wide policies to prevent the postulated attacks, DROIDGUARD parses and transforms them from models generated in relational logic to a set of configurations directly amenable to efficient policy enforcement. Our policy enforcer (APE) then instruments each vulnerable app's bytecode to dynamically intercept event messages, check them against generated policies, and possibly inhibits their executions if violating any such policies. As such, to the best of our knowledge, DROIDGUARD is the first approach capable of detecting and protecting Android systems against zero-day inter-app attacks.

In the following three sections, we describe the details of each component in turn.

5 AME: Android Model Extraction

In order to automatically analyze vulnerabilities, we first need to extract a model of each app's behavior to reason about its security properties. This section first defines the model we extract for each app, and then describes the extraction process.

Definition 1. *A model for an Android application is a tuple* $App =< Cmps, Intents, IFltrs, Perms, Paths >$*, where*

- *Cmps* is a set of components, where each component $c \in Cmps$ has a set of Intent messages $intents(c) \subseteq Intents$, a set of Intent filters $ifltrs(c) \subseteq IFltrs$, a set of permissions $perms(c) \subseteq Perms$ required to access the component c, and a set of sensitive paths $paths(c) \subseteq Paths$.
- *Intents* is a set of event messages that can be used for both inter- and intra-application communications.
- *IFltrs* is a set of Intent filters, where each Intent filter $ifilter \in IFltrs$ is attached to a component $c \in Cmps$.
- *Perms* is a union of required and enforced permissions, $Perm = Perms_{Req} \cup Perms_{Enf}$, where $Perms_{Req}$ specifies the permissions to which the application needs to have access to run properly and $Perms_{Enf}$ specifies the permissions required to access components of the application under consideration. We let the set of permissions *actually* used within a component c as $perm_{Used}(c) \subseteq Perms_{Req}$.
- *Paths* is a finite set of sensitive data flows; each data flow belongs to a component $c \in Cmps$ and is represented as a tuple $< Source, Sink >$, where *Source* represents a sensitive data (e.g., the device ID) and *Sink* represents a method that may leak data, such as sending text messages.

Architecture Extraction. To obtain an app model, AME first examines the app manifest file to capture the high-level architectural information, including the components (*Cmps*) comprising the app, permissions that the app requires ($Perms_{Req}$), and the enforced permissions ($Perms_{Enf}$) that the other apps must have in order to interact with the app components. AME also identifies public interfaces exposed by each application, which are essentially entry points defined in the manifest file through Intent Filters (*IFltrs*) of components.

Intent Extraction. The next step of model extraction involves an inter-procedural data flow analysis [15], to track the Intents and Intent Filters that are declared in code, rather than the manifest file, as well as their properties. Each Intent ($i \in Intents$) belongs to one particular component sending it ($sender(i) \in Cmps$), may have one recipient component ($component(i) \in Cmps$), and three sets of $action(i)$, $data(i)$ and $categories(i)$ specifying the general action to be performed in the recipient component, additional information about the data to be processed by the action, and the kind of component that should handle i, respectively.

Similar to Intents, each Intent filter ($ifilter$) has a non-empty set of $actions(ifilter)$ and two sets of $data(ifilter)$ and $categories(ifilter)$. Note that Intent filters for components of type Service and Activity must be declared in their manifest; for Broadcast Receivers, though, either in the manifest or at runtime.

To resolve the values associated with the retrieved attributes (e.g., the Intent action) AME uses string constant propagation [19], which provides a suitable solution since, by convention, Android apps use constant strings to define these values. In case a property is disambiguated to more than one value (e.g., due to a conditional assignment), AME generates a separate entity for each of these values, as they contribute different exposure surfaces or event messages in case of Intent filters and Intents, respectively. AME handles aliasing through performing *on-demand alias analysis* [52]. More specifically, for each attribute that is assigned to a heap variable, the backward analysis finds its aliases and updates the set of its captured values accordingly.

Path Extraction. AME analyzes the app using a static taint analysis to track sensitive data flow tuples $< Source, Sink >$. To achieve a high precision in data flow analysis, our approach is flow-, field-, and context-sensitive [11], meaning that our analysis distinguishes a variable's values between different program points, distinguishes between different fields of a heap object, and that in analysis of method calls is sensitive to their calling contexts, respectively. In the interest of scalability, DROIDGUARD's analysis, however, is not path-sensitive. The results (cf. Sec. 9) though indicate no significant imprecision caused by path-insensitivity in the context of Android vulnerability analysis.

AME uses a set of most frequently used source and sink Android API methods from the literature [44], identified through the use of machine-learning techniques. This set has a recall and precision of over 90%, thus is sufficient for our testing and evaluation. To further detect those paths traversing through different components, we adapted this set by identifying source and sink methods corresponding to inter-component communication.

Figure 3 shows two instances of sensitive data paths in models derived for our running example. The first one starts from node Ⓜ, where *FindLocationActivity* retrieves device location and sends it over an implicit Intent, leading to a sensitive path from source method `getLastKnownLocation` to sink method `startActivity`. Another path is also similarly identifiable within the *TelephonyActivity* component. The involving nodes, paths and components are highlighted by red color in Figure 3. These identified paths are later used in the ASE module to detect data leaks vulnerabilities, and thereby to generate respective policies preventing their potential exploits.

Permission Extraction. To ensure the permission policies are preserved during an inter-component communication, one should compare the granted permissions of the caller component against the enforced permissions at the callee component side. Therefore, the permissions actually used by each component should be determined. While we already identified the coarse-grained permissions specified in the manifest file, AME analyzes permission checks throughout the code to identify those controlling access to particular aspects of a component (e.g., recall *hasPermission* method of Listing 2). In doing so, it relies on API permission maps available in the literature, and in particular the PScout permission map [12], one of the most recently updated and comprehensive permission maps available for the Android framework. API permission maps specify mappings between Android API calls/Intents and the permissions required to perform those calls.

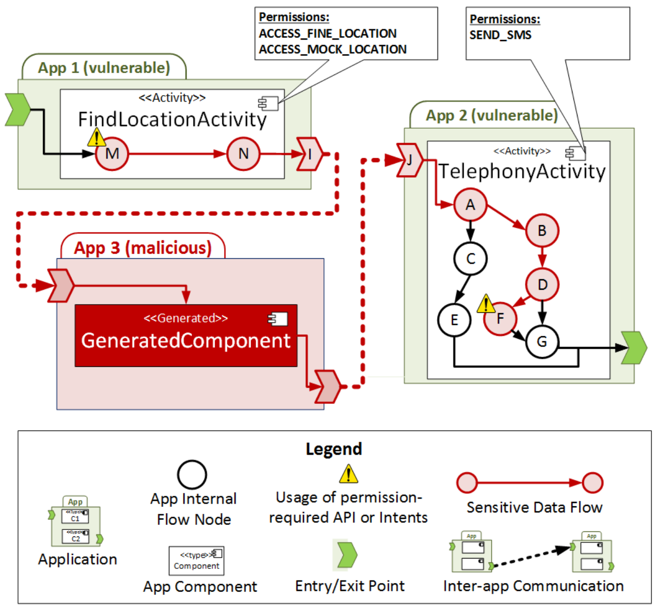

Figure 3: Extracted models of the apps described in listings 1 and 2. The malicious app's (App3) model is not extracted by Model Extractor, rather its specification is automatically generated by our synthesis engine.

As shown in Figure 3, a node could be directly tagged as *permission-required* node (shown by ⚠ sign), or transitively tagged by tracking the call chains. To find the transitive permission tag, AME performs backward reachability analysis starting from the permission-required node. The tagged permission are propagated from all children to their parent nodes, until reaching to the root nodes. In case an entry point node of a component is tagged by a permission, it will be added to the list of used permissions of that component, $perm_{Used}(c)$. For instance, as shown in Figure 3, *FindLocationActivity* components requires `ACCESS_FINE_LOCATION` due to calling GPS APIs, and *TelephonyActivity* component needs `SEND_SMS` as it uses text messaging service.

6 ASE: Vulnerability Analysis and Policy Synthesis Engine

We now show that our ideas for automated synthesis of exploit specifications can be reduced to practice. The insight that enabled such synthesis was that we could interpret the synthesis problem as the dual of formal verification. Given a system specification S, a model M, and a property P, formal verification asserts whether M satisfies the property P under S. Whereas the synthesis challenge is given a system specification S and a property P, generate a model M satisfying the property P under system S. M is an instance model of S that satisfies P.

This observation enables leveraging verification techniques to solve synthesis problems. As shown in Figure 4, we can view the bundle of app specifications, S_a, and the framework specification, S_f, collectively as system S and a compositional security issue as property P, and model them as a set of constraints. The problem then becomes to generate a candidate set of violation scenarios, M, that satisfies the space of constraints: $M \models S_f \wedge S_a \wedge P$. Our approach is thus based on a reduction of the synthesis problem into a constraint-solving problem represented in relational logic (i.e., Alloy). Alloy is a formal modeling language with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [36], making it an appropriate language for declarative specification of both applications and properties to be checked (i.e., possible exploits).

The formulation of the synthesis problem in Alloy consists of three parts: (1) a fixed set of signatures and facts describing the Android application fundamentals (e.g., application, component, Intent, etc.) and the constraints that every application must obey. Technically speaking, this module can be considered as a meta-model for Android applications; (2) a separate Alloy module for each app modeling various parts of an Android app extracted from its APK file. The automatically extracted model for each app relies on the Android framework specification module (the first item above); and (3) a set of signatures used to reify inter-component vulnerabilities in Android, such as privilege escalation. The rest of this section details each part in turn.

Formal Model of Android Framework. Listing 3 shows (part of) the Alloy code describing the meta-model for Android application models (Appendix A provides a brief overview of Alloy). For example, note the signatures `Component` and `Intent`[4]. A component belongs to exactly one application, and may have any number of `IntentFilters`—each one describing a different interface (capability) of the component—and a set of permissions required to access the component. The `paths` field then indicates information flows between permission domains. We define the source and destination of a path based on canonical permission-required resources identified by Holavanalli et al. for Android applications [32]. Examples of such resources are NETWORK, IMEI, and SDCARD. Thirteen permission-required resources are identified as source, and five resources as destination, of a sensitive data flow path. The IPC mechanism augments both source and destination sets.

[4]In the Alloy language, signatures (sig) provide the vocabulary of a model by defining the basic types of elements and the relationships between them.

```
abstract sig Component{
  app: one Application,
  intentFilters: set IntentFilter,
  permissions: set Permission,
  paths: set DetailedPath }
abstract sig IntentFilter{
  actions: some Action,
  data: set Data,
  categories: set Category }
fact IFandComponent{
  all i:IntentFilter|
    one i.~intentFilters }
fact NoIFforProviders{
  no i:IntentFilter|
    i.~intentFilters in Provider }
abstract sig Intent{
  sender: one Component,
  component: lone Component,
  action: lone Action,
  categories: set Category,
  data: set Data,
  extra: lone Resource }
```

Listing 3: Excerpts from the meta-model for Android application models in Alloy.

The fact[5] `IFandComponent` specifies that each IntentFilter belongs to exactly one `Component`, and the fact `NoIFforProviders` specifies that out of four core component types, only three of them can define IntentFilters; no IntentFilter can be defined for `Content Provider` components.

An Intent belongs to one particular component sending it, and may have one recipient component. Each Intent also includes three sets of `action, data` and `categories`. These three elements are used to determine to which component an *implicit Intent*—one that does not specify any recipient component—should be delivered. Each of these elements corresponds to a test, in which the Intent's element is matched against that of the IntentFilter. An IntentFilter may have more actions, data, and categories than the Intent, but it cannot contain less. The `extra` field indicates the type of data carried by the Intent in terms of permission-required resources it is obtained from.

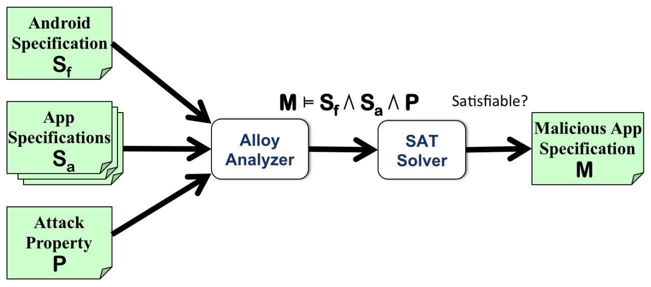
Android Specification S_f
App Specifications S_a
Attack Property P
$M \models S_f \wedge S_a \wedge P$
Satisfiable?
Alloy Analyzer
SAT Solver
Malicious App Specification M

Figure 4: Automated synthesis of possible exploit specifications.

Formal Model of Apps. Listing 4 partially shows the generated specifications for the apps shown in Listings 1 and 2. Each app model starts by importing the *androidDeclaration* module (cf. Listing 3). Consider the extracted models for our running example (cf. Fig. 3) and the corresponding generated Alloy models. There is a dataflow from the node *M*, where the sensitive GPS data is retrieved, to the Intent *I*. The component thus contains

[5]In Alloy, facts define constraints that must always hold.

```
1  (a) App1 model
2  open androidDeclaration
3  ...
4  one sig FindLocationActivity extends Activity{}{
5    app in App1
6    no intentFilters
7    paths = path_FindLocationActivity_1
8    permissions = ACCESS_FINE_LOCATION }
9  one sig path_FindLocationActivity_1 extends Path{}{
10   source = LOCATION
11   sink = IPC }
12 one sig intent2_1 extends Intent{}{
13   sender = LocActivity
14   no component
15   action=showLoc
16   categories= DEFAULT
17   data = NoMimeNoScheme
18   extra= LOCATION }
19 (b) App2 model
20 one sig TelephonyActivity extends Activity{}{
21   app in App2
22   intentFilter = IntentFilter1
23   paths = path_TelephonyActivity_1
24   no permissions }
25 one sig path_TelephonyActivity_1 extends Path{}{
26   source = IPC
27   sink = SMS }
```

Listing 4: Excerpts from generated specifications for (a) App1 (Listing 1) and (b) App2 (Listing 2).

```
1  sig GeneratedActivityLaunch{
2    disj launchedCmp,malCmp: one Component,
3    malIntent: Intent  }{
4    malIntent.sender = malCmp
5    launchedCmp in setExplicitIntent[malIntent]
6    no launchedCmp.app & malCmp.app
7    launchedCmp.app in device.apps
8    not (malCmp.app in device.apps)
9    some launchedCmp.paths && launchedCmp.paths.source
         = IPC
10   some malIntent.extra
11   malCmp in Activity
12 }
```

Listing 5: Alloy specifications of Activity Launch vulnerability in Android.

a sensitive path (`path_FindLocationActivity_1`), and the `extra` field of the Intent in the generated Alloy model (line 18) is accordingly set. On the other hand, there is a data-flow, started from the J IntentFilter and reaches to node F, which uses the data in the body of a text message. The `path` field of the *TelephonyActivity* in the generated Alloy model (line 23, 25–27) thus reflects this path. Note that this component does not enforce any access permission neither in the manifest file nor in the source code (line 24).

Formal Model of Vulnerabilities. To provide a basis for precise analysis of app bundles against inter-app vulnerabilities and further to automatically generate possible scenarios of their occurrence given particular conditions of each bundle, we designed specific Alloy signatures. As a concrete example, we illustrate the semantics of one of these vulnerabilities in the following. The others are evaluated similarly.

Listing 5 presents the `GeneratedActivitylaunch` signature along with its *signature fact* that specifies the elements involved in, and the semantics of, an Activity launch exploit, respectively. In short, a malicious component (`malCmp`) can launch an Activity by sending an Intent (`malIntent`) to an exported component (`launchedCmp`) that is not expecting Intents from that component. According to line 9, the *launchedCmp* component has a path from the exported interface to a permission-required resource. It, thus, may leak information or perform unauthorized tasks, depending on the functionalities exposed by the victim component.

Generating possible exploit scenarios. We run the modules defined above with a command that tries to satisfy the vulnerabilities signature facts. Note that Alloy analysis must be done within a given scope, which specifies an upper bound for, or an exact, number of instances per element signature. In our case, the exact scope of each element, such as Application and Activity, required to instantiate each vulnerability is automatically derived from the specification.

If an instance is found, DROIDGUARD reports it along with the information useful in finding the root cause of the violation, from which fine-grained security policies are then derived for the given system. Given our running example, the analyzer automatically generates the following scenario:

```
... // omitted details of model instances
GeneratedInfoLeak={GeneratedInfoLeak$0}
|_components={vulnerableCmp$1,vulnerableCmp$2,malCmp}
 |_vulnerableCmp$1={App1/FindLocationActivity}
 | |_vulnerableIntent$1={FindLocationActivity/intent2_1}
 |_vulnerableComponent$2={App2/TelephonyActivity}
 |_malCmp={Application$0/Activity$0}
  |_malIntent$1={Activity$0/Intent$0}
   |_sender={Application$0/Activity$0}
   |_component={App1/TelephonyActivity}
   |_action={sendSMS}
   |_extra = {LOCATION}
```

It essentially states the scenario represented in Figure 1, in which a postulated malicious component, here the generated `Application$0/Activity$0` component, can send the device location data captured from a vulnerable Intent, `intent2_1`, to the desirable phone number via an explicit Intent, `Activity$0/Intent$0`, sent to the `App2/TelephonyActivity` component that is vulnerable to Activity launch.

The next section describes how we can prevent occurrence of such vulnerability exploits through generation and enforcement of respective policies.

7 APE: Android Policy Enforcer

In the implementation of APE, we faced two possible alternatives: (1) modify the Android OS to enforce the policies, or (2) modify the apps through injection of policy enforcement logic into the APK file. We chose the latter approach, as it allows DROIDGUARD to be used on an unmodified version of Android, thereby making it widely applicable and practical for use by many.

Similar to a conventional access control model [46], our approach is comprised of two elements: *policy de-*

cision point (PDP)—the entity which evaluates access requests against a policy, and *policy enforcement point (PEP)*—the entity which intercepts the request to a resource, makes a decision request to the PDP, and acts on the received decision. The protected resources in our research are mainly Android APIs that can result in IPC calls. The PDP is realized by instrumenting every application identified as vulnerable with a module that relies on the synthesized policies for preventing or allowing IPC access. The PEP in our case corresponds to the instrumentation of code to dynamically intercept event messages. More specifically, each IPC method in an app's APK file (e.g., `startActivity(Intent)`) is replaced with a guarded operation that checks whether the operation should proceed (e.g., `Intent` to be delivered to its destination) by calling the PDP.

Sensitive operations in Android applications, such as sending SMS messages, are conducted through calls to Android API methods. PEP wraps these operations and uses PDP to check whether they are allowed to run or not. Whenever an application is about to run a sensitive operation, it is checked against the synthesized policies. The respective application is then allowed to perform the given operation as long as it conforms to such policies. Otherwise, the PDP prompts the user for consent along with the information that would help the user in making a decision, including the description of security threat as well as the name and parameters of the intercepted event. Should the user refuse, the application skips the given operation and continues with running the subsequent one. As IPC mechanisms in Android are essentially performed by asynchronous API calls, inhibiting them implies that no response for the event is ever received, without causing unexpected crashes. Of course, preventing IPC calls would naturally force the app to operate in a degraded mode.

Continuing with our running example, given the generated scenario, App2 will be integrated with the following policy: *every attempt of sending device LOCATION data through SMS must be manually approved by the user.* Observe that each app, such as App2 can, and in this case would, be guarded against more than one policy at the same time. Indeed, App1 and App2 would also be instrumented with policies generated regarding the Intent hijacking and Activity Launch, respectively.

8 Tool Implementation

We have implemented DROIDGUARD as a publicly available tool [4]. We have built our static analysis capability on top of the Soot [53] framework. We used Flowdroid for intra-component taint analysis [11], and extended it to improve precision of analysis especially to support some complicated ICC methods, such as `startActivityForResult`. The prototype implementation of DROIDGUARD only requires the APK files—not the original source code—which is important, of course, for running it over non-open source apps. The translation of captured app models into the Alloy language are implemented using FreeMarker template engine [6]. The core components of our analysis and synthesis model are embedded in a relational logic language, i.e., Alloy [36]. As a back-end analysis engine, DROIDGUARD relies on Aluminum [42], a recently developed principled scenario explorer that generates only minimal scenarios for specifications axiomatized in Alloy. Lastly, our policy enforcer (cf. APE module) leverages the Soot framework [53] to inject instrumented code—dynamically preventing event messages violating synthesized policies—into each vulnerable application.

9 Evaluation

In this section, we present the experimental evaluation of DROIDGUARD. Our evaluation addresses the following research questions:

RQ1. What is the overall accuracy of DROIDGUARD in detecting IPC (i.e., both inter-component and inter-application) vulnerabilities compared to other state-of-the-art techniques?

RQ2. How well does DROIDGUARD perform in practice? Can it find security exploits and synthesize their corresponding protection policies in real-world applications?

RQ3. What is the performance of DROIDGUARD's analysis implemented atop static analyzers and SAT solving technologies?

RQ4. How well does DROIDGUARD's policy enforcement perform compared with existing tools?

9.1 Results for RQ1 (Accuracy)

To evaluate the effectiveness and accuracy of our analysis technique and compare it against the other static analysis tools, we used the DroidBench [3] and ICC-Bench [8] suites of benchmarks, two sets of Android applications containing IPC based privacy leaks for which all vulnerabilities are known in advance—establishing a ground truth. These test cases comprise the most frequently used IPC methods found in Google Play apps. The benchmark apps also include unreachable, yet vulnerable components; reported vulnerabilities that involve such unreachable components are thus considered as false warnings. Using the apps in this benchmark, which is developed by other research groups, we have attempted to eliminate internal threats to the validity of our results. Further, using the same benchmark apps as prior research allows us to compare our results against them.

We compared DROIDGUARD with existing tools targeted at ICC vulnerability detection, namely DidFail [37] and AmanDroid [55]. COVERT [14] only targets a specific type of inter-app vulnerability, i.e. privilege escalation. We excluded COVERT from our comparison, as all of the apps in DroidBench and ICC-Bench are examples of information leakage type of vulnerabilities that COVERT cannot detect. We also tried to run IccTA [38], another tool intended to identify inter-app vulnerabilities, but faced technical difficulties. The tool terminated with error while capturing ICC links. This issue has also been reported by others [9]. Though we have been in contact with the authors, we have not been unable to fix it so far.

To compare the accuracy of selected tools, we measured precision, recall, and F-measure as follows:

Precision is the percentage of those vulnerabilities detected by the tool that were also classified as vulnerabilities by the benchmark: $\frac{TP}{TP+FP}$

Recall is the percentage of real vulnerabilities that the tool finds: $\frac{TP}{TP+FN}$

F-measure is the harmonic mean of precision and recall: $\frac{2*Precision*Recall}{Precision+Recall}$

where TP (true positive), FP (false positive), and FN (false negative) represent the number of vulnerabilities that are correctly detected, falsely reported, and missed.

Table 1 summarizes the results of our experiments for evaluating the accuracy of DROIDGUARD in detecting IPC vulnerabilities compared to other state-of-the-art techniques. DROIDGUARD succeeds in detecting all 23 known vulnerabilities in DroidBench benchmarks, and 7 vulnerabilities out of 9 in ICC-Bench suite. It correctly finds both cases of privacy leak in *bindService4* and *startActivityForResults4*. It also correctly ignores two cases where there are no leaks, since the code harboring those vulnerabilities is not reachable, i.e., *startActivity*{*4*,*5*}. The only missed vulnerabilities are the ones that are caused by dynamic registration of Broadcast Receivers, which is not handled by DROIDGUARD's model extractor.

In addition to missing the vulnerabilities in the bound services, AmanDroid is unable to examine Content Providers for security analysis. DidFail does even worse. Based on the results, DidFail found only the vulnerabilities caused by implicit Intents, missing vulnerabilities that are due to explicit Intents, such as information leak.

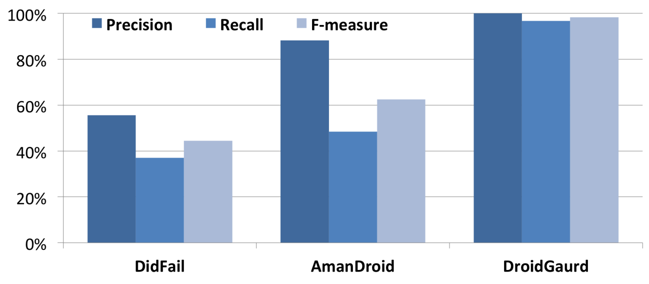

Figure 5: ICC Benchmark Comparison Results.

Table 1: Comparison between DROIDGUARD, DidFail, and AmanDroid. TP, FP and FN are represented by symbols ☑, ☒, ☐, respectively. (X#) indicates the number # of detected instances for the corresponding symbol X.

	Test Case	DidFail	AmanDroid	DroidGuard
DroidBench2	ICC_bindService1	☒☐	☐	☑
	ICC_bindService2	☐	☐	☑
	ICC_bindService3	☐	☐	☑
	ICC_bindService4	☒(☐2)	(☐2)	(☑2)
	ICC_sendBroadcast1	☑	☑	☑
	ICC_startActivity1	☐	☑	☑
	ICC_startActivity2	☐	☑	☑
	ICC_startActivity3	☐	☑	☑
	ICC_startActivity4	☒		
	ICC_startActivity5	(☒2)		
	ICC_startActivityForResult1	☐	☑	☑
	ICC_startActivityForResult2	☐	☐	☑
	ICC_startActivityForResult3	☐	☐☒	☑
	ICC_startActivityForResult4	(☐2)	☑☒☐	(☑2)
	ICC_startService1	☒☐	☑	☑
	ICC_startService2	☒☐	☐	☑
	ICC_delete1	☐	☐	☑
	ICC_insert1	☐	☐	☑
	ICC_query1	☐	☐	☑
	ICC_update1	☐	☐	☑
	IAC_startActivity1	☑☒	☐	☑
	IAC_startService1	☑	☐	☑
	IAC_sendBroadcast1	☑	☐	☑
ICC-Bench	Explicit_Src_Sink	☐	☑	☑
	Implicit_Action	☑	☑	☑
	Implicit_Category	☑	☑	☑
	Implicit_Data1	☑	☑	☑
	Implicit_Data2	☑	☑	☑
	Implicit_Mix1	☑	☑	☑
	Implicit_Mix2	☑	☑	☑
	DynRegisteredReceiver1	☐	☑	☐
	DynRegisteredReceiver2	☐	☐	☐

The precision, recall and F-measure for the benchmark apps are depicted in Figure 5. The results show that DROIDGUARD outperforms the other two tools in terms of both precision and recall.

9.2 Results for RQ2 (DROIDGUARD and Real-World Apps)

To evaluate the implications of our tool in practice, we collected 4,000 apps from the following four different sources:

Google Play [7]. This repository serves as the official Android app store. Our Google play collection consists of 600 randomly selected and 1,000 most popular free apps in the market.

F-Droid [5]. This is a software repository that contains free and open source Android apps. Our collection includes 1,100 apps from this Android market.

Malgenome [58]. This repository contains malware samples that cover the majority of existing Android malware families. Our collection includes all (about 1,200) apps in this repository.

Table 2: Vulnerability Types and Samples of detected apps for each type.

Vulnerability Types	Vulnerable Apps
Activity/Service Launch	**Barcoder:** Exposes bill payment service by receiving an implicit Intent.
Intent Hijack	**Hesabdar:** Sends users accounts as an implicit Intents, could be received by unauthorized apps.
Information Leakage	**OwnCloud:** Leaks cloud account information to the Android shared log.
Privilege Escalation	**Ermete SMS:** Gives out the SMS permission to other apps who may not have it.

Bazaar [1]. This website is a third-party Android market. We collected 100 popular apps from this repository, distinguished from apps downloaded from Google Play and F-Droid.

We partitioned the subject systems into 80 non-overlapping bundles, each comprised of 50 apps, simulating a collection of apps installed on an end-user device. The bundles enabled us to perform several independent experiments. We have made the list of apps in each bundle available to the reader [4]. Discovered vulnerabilities can be categorized into four classes, shown in the first column of Table 2. Out of 4,000 apps, DROIDGUARD identified 97 apps vulnerable to Intent hijack, 124 apps to Activity/Service launch, 128 apps to inter-component sensitive information leakage, and 36 apps to privilege escalation. We then manually inspected the DROIDGUARD's results to assess its utility in practice. In the following, we describe some of our findings. To avoid leaking previously unknown vulnerabilities, we only disclose a subset of those that we have had the opportunity to bring to the app developers' attention.

Activity/Service Launch. *Barcoder* is a barcode scanner app that scans bills using the phone's camera, and enables users to pay them through an SMS service. It also stores the user's bank account information, later used in paying the bills. Given details of a bill as payload of an input Intent, the *InquiryActivity* component of this app pays it through SMS service. This component exposes an unprotected Intent filter that can be exploited by a malicious app for making an unauthorized payment.

Intent Hijack. *Hesabdar* is an accounting app for personal use and money transaction that, among other things, manages account transactions and provides a temporal report of the transaction history. One of its components handles user account information and sends the information as payload of an implicit Intent to another component. When a component sends an implicit Intent, there is no guarantee that it will be received by the intended recipient. A malicious application can intercept an implicit Intent simply by declaring an Intent filter with all of the actions, data, and categories listed in the Intent, thus stealing sensitive account information by retrieving the data from the Intent.

Information Leakage. *OwnCloud* provides cloud-based file synchronization services to the user. By creating an account on the back-end server, user can sync selected files on the device and access synced files to browse, manage, and share. Our study indicates that OwnCloud app is vulnerable to leak sensitive information to other apps. One of its components obtains the account information and through a chain of Intent message passing, eventually logs the account information in an unprotected area of the memory card, which can be read by any other app on the device.

Table 3: Experiments performance statistics.

Components	Intents	Intent Filters	Time (sec) Construction	Analysis
313	322	148	260	57

Privilege Escalation. *Ermete SMS* is a text messaging application with *WRITE_SMS* permission. Upon receiving an Intent, the *ComposeActivity* component of this app extracts the payload of the given Intent, and sends it via text message to a number also specified in the payload, without checking the permission of the sender. This vulnerable component, thus, provides the *WRITE_SMS* permission to all other apps that may not have it.

9.3 Results for RQ3 (Performance and Timing)

The next evaluation criteria are the performance benchmarks of static model extraction and formal analysis and synthesis activities. We used a PC with an Intel Core i7 2.4 GHz CPU processor and 4 GB of main memory, and leveraged Sat4J as the SAT solver during the experiments.

Figure 6 presents the time taken by DROIDGUARD to extract app specifications for 4,000 real-world apps. This measurement is done on the data-sets collected from 4 repositories: Google Play, F-Droid, Malgenome, and Bazaar. The scatter plot shows both the analysis time and the app size. According to the results, our approach statically analyzes 95% of apps in less than two minutes. As our approach for model extraction analyzes each app independently, the total static analysis time scales linearly with the size of the apps.

Table 3 shows the average time involved in compositional analysis and synthesis of policies for a set of apps. The first three columns represent the average number of Components, Intents, and Intent filters within each analyzed bundle. The next two columns represent the time spent on transforming the Alloy models into 3-SAT clauses, and in SAT solving to find the space of solutions for each bundle. The timing results show that on average DROIDGUARD is able to analyze bundles of apps containing hundreds of components in the order of a few minutes (on an ordinary laptop), confirming that the proposed technology based on a lightweight formal analyzer is feasible.

Finally, we compared scalability of DROIDGUARD with other tools that support analysis of inter-app vulnerabilities, namely DidFail and AmanDroid. Figure 7 compares the analysis time taken by each of these tools. We chose configuration of apps from the already reported repositories (cf. RQ2) with the number of components specified on the x axis. As illustrated in the diagram, the analysis time by AmanDroid scales exponentially, and for a bundle of just 8 components it exceeds 10 minutes. DidFail performs better in terms of time, but fails to analyze apps with more than 30 components, with error in its transformation phase. The results show that DROIDGUARD outperforms the two other tools in terms of scalability.

9.4 Results for RQ4 (policy enforcement)

The last evaluation criteria are the performance benchmarks of DROIDGUARD's policy enforcement. Recall from Section 7 that DROIDGUARD instruments each vulnerable app with a PDP, a module that implements policy decision making. We measured the overhead introduced by APE due to bytecode instrumentation of apps with PDP. This module accounts for 522 instructions. On average, the overhead of instrumented instructions for the subject systems we measured is less than one percent, and more specifically 0.22%.

To measure the runtime overhead required for PEP (i.e., policy enforcement), we tested an instrumented application on a Nexus 1 phone (Android version 2.3.7). Our benchmark application repeats performing an IPC operation, namely *startService*, for 100 times. We have handled uncontrollable factors in our experiments by repeating the experiments 33 times and reporting the results using their 95% confidence intervals. Overall, the execution time overhead incurred by APE for policy enforcement is 4.94% $\pm$ 1.06%, making the effect on user experience negligible.

We also tried to compare dynamic policy enforcement feature of DROIDGUARD with other state-of-the-art tools, namely AppFence [34], AppGuard [13], Aurasium [57], DroidForce [45], and Pegasus [17]. Out of these five tools, AppGuard, Aurasium and DroidForce were publicly available. For the other tools, we contacted the respective authors. Unfortunately, the authors of Pegasus stated that it is not publicly available, and we never heard back from AppFence.

We tested the available tools against the benchmark apps of DroidBench2.0 [3]. Aurasium's enforcement is based on repackaging the application and attaching policy logic to it. However, the tool was not able to repackage any of the DroidBench apps and terminated with an internal error. We contacted the authors, but never heard back. To run DroidForce tool, we developed a set of appropriate policies for preventing ICC leaks in benchmark apps. However, DroidForce was not able to correctly enforce the policies. The issue is confirmed by the authors. We installed AppGuard app on a phone, and it correctly instrumented the benchmark apps, which allows us to revoke permissions after app-installation time. However, policies that can be enforced by AppGuard are limited to the permissions enabling access to sensitive sources and sinks, rather than narrowly targeted policies preventing, e.g. certain vulnerable inter-component communications, as in DROIDGUARD. In conclusion, while there is a number of policy-enforcement tools for Android available in the scientific literature, we were unable to successfully fix ICC vulnerabilities in the benchmark apps.

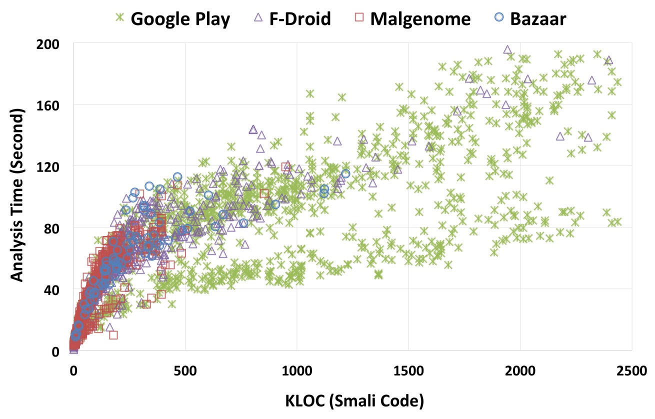

Figure 6: Scatter plot representing analysis time for model extraction of Android apps.

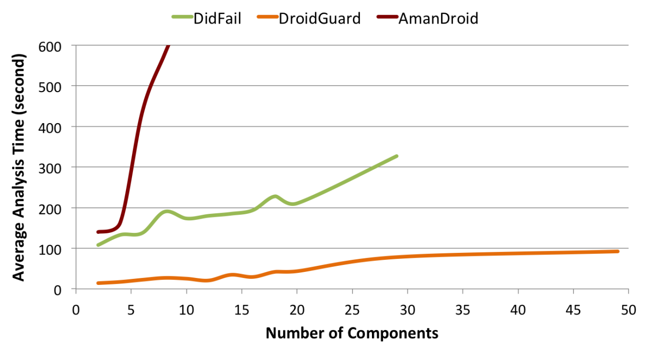

Figure 7: Performance Comparison Results. To improve the readability, the y-axis is limited to 10 minutes (600 seconds), which intercepts Amandoird line at 8-components point. DidFail was unable to analyze apps with more than 30 components.

10 Related Work

Mobile security issues have received a lot of attention recently. Here, we provide a discussion of the related efforts in light of our research.

A large body of work [18,23,28–30,41,43,59] focuses on performing program analysis over Android applications for security. Chin et al. [18] studied security challenges of Android communication, and developed ComDroid to detect those vulnerabilities through static analysis of each app. Octeau et al. [43] developed Epicc for analysis of Intent properties—except data scheme—through inter-procedural data flow analysis. FlowDroid [11] introduces a precise approach for static taint flow analysis in the context of each application component. CHEX [39] also takes a static method to detect component hijacking vulnerabilities within an app. These research efforts, like many others we studied, are mainly focused on Intent and component analysis of one application. DROIDGUARD's analysis, however, goes far beyond single application analysis, and enables synthesis of policies targeting the overall security posture of a system, greatly increasing the scope of vulnerability analysis.

The other, and perhaps more closely related, line of research focuses on ICC analysis [14,37,38,55,56,59]. DidFail [37] introduces an approach for tracking data flows between Android components. It leverages Epicc for Intent analysis, but consequently shares Epicc's limitation of not covering data scheme, which negatively affects the precision of this approach in inter-component path matching. Moreover, it does not generate nor enforce system-specific policies, as performed by DROIDGUARD. IccTA, similarly, leverages an intent resolution analysis to identify inter-component privacy leaks [38]. IccTA's approach for inter-component taint analysis is based on a pre-processing step connecting Android components through code instrumentation, which improves accuracy of the results but may also cause scalability issues. Amandroid also tackles Android ICC-based privacy leaks [55]. It does not support one of the four types of Android components, i.e., Content Provider, nor complicated ICC methods, like startActivityForResult. Along the same line, COVERT [14] presents an approach for compositional analysis of Android inter-app vulnerabilities. While this work is concerned with the analysis of permission leakage between Android apps, it does not really address the problem that we are addressing, namely the automated synthesis and dynamic enforcement of system-specific policies.

The other relevant thrust of research has focused on policy enforcement [13,17,34,45,47,54,57]. Kirin [24] extends the application installer component of Android's middleware to check the permissions requested by applications against a set of security rules. These predefined rules are aimed to prevent unsafe combination of permissions that may lead to insecure data flows. Our work differs in that it generates system-specific, fine-grain policies for a given system, rather than relying on general-purpose policies defined based only on coarse-grain permissions. Moreover, DROIDGUARD is more precise as it dynamically analyzes policy violations against flows that actually occur at run-time.

Along the same line, Kynoid [47] enforces policies at runtime through performing a dynamic taint analysis over a modified version of Dalvik VM. This approach, similar to many of the previously proposed solutions [22–24, 26], requires changes to the Android. Our approach, in contrast, requires no platform modifications. Rasthofer et al. proposed DroidForce [45], in which each sensitive operation should be approved by a central app before being executed by individual apps. This centralized architecture, however, implies both performance and security deficiencies. More recently, DeepDroid [54] presents a policy enforcement scheme based on dynamic memory instrumentation of system processes on Android devices. However, it depends on undocumented internal architecture of Android framework and its system resources which may change in future versions without notice.

All the enforcement techniques we studied rely on policies developed by users, whereas DROIDGUARD is geared towards the application of formal techniques to synthesize such policies through compositional analysis of Android apps. Our work can complement their research by providing highly precise synthesized policies to relieve their users of responsibility of manual policy development.

Finally, constraint solving for synthesis and analysis has increasingly been used in a variety of domains [10, 31, 49–51]. These research efforts share with ours the common insight of using the state-of-the-art constraint solving for synthesis. Different from all these techniques, DROIDGUARD tackles the automated detection and mitigation of inter-app security vulnerabilities in Android, by synthesizing Android-specific security policies. It thus relieves the tedium and errors associated with their manual development. To the best of our knowledge, DROIDGUARD is the first formally-precise technique for automated synthesis and dynamic enforcement of Android security policies.

11 Concluding Remarks and Limitations

This paper presents a novel approach for automatic synthesis and enforcement of security policies, allowing the end-users to safeguard the apps installed on their device from inter-app vulnerabilities. The approach, realized in a tool, called DROIDGUARD, combines static program analysis with lightweight formal methods to automatically infer security-relevant properties from a bundle of apps. It then uses a constraint solver to synthesize possible security exploits, from which fine-grained security policies are derived and automatically enforced to protect a given device. The results from experiments in the context of thousands of real-world apps corroborates DROIDGUARD's ability in finding previously unknown vulnerable apps as well as preventing their exploitation.

Our approach has a few limitations. Current implementation of DROIDGUARD mainly instruments API calls at the bytecode level. It thus might miss methods executed in native libraries accessed via Java Native Interface (JNI), or from external sources that are dynamically loaded. It has been shown that only about 4.52% of the apps on the market contain native code [60]. Supporting these additional sources of vulnerability entails extensions to our static program analysis and instrumentation approach to support native libraries. Reasoning about dynamically loaded code is not possible through static analysis, and thus, an additional avenue of future work is leveraging dynamic analysis techniques, such as TaintDroid [23] and EvoDroid [40], that would allow us to extract additional behaviors that might be latent in apps.

References

[1] Bazaar. http://cafebazaar.ir/.

[2] Dalvik - code and documentation. http://code.google.com/p/dalvik/.

[3] Droidbench2.0. http://github.com/secure-software-engineering/DroidBench/tree/iccta/apk.

[4] Droidguard. Redacted due to double blind submission requirement.

[5] F-droid. https://f-droid.org/.

[6] Freemarker java template engine. http://freemarker.org/.

[7] Google play market. http://play.google.com/store/apps/.

[8] Iccbench. https://github.com/fgwei/ICC-Bench/tree/master/apks.

[9] Iccta tool on github, reported issues. https://github.com/lilicoding/soot-infoflow-android-iccta/issues/7.

[10] Akhawe, D., Barth, A., Lam, P., Mitchell, J., and Song, D. Towards a formal foundation of web security. In *Proceedings of the IEEE Computer Security Foundations Symposium* (2010).

[11] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)* (2014).

[12] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: Analyzing the android permission specification. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012).

[13] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard–enforcing user requirements on android apps. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 543–548.

[14] Bagheri, H., Sadeghi, A., Garcia, J., and Malek, S. Covert: Compositional analysis of android inter-app vulnerabilities. Tech. Rep. GMU-CS-TR-2015-1, Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030-4444 USA, 2015.

[15] Bodden, E. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis* (2012), ACM, pp. 3–8.

[16] Bugiel, S., David, L., Dmitrienko, A. Fischer, T., Sadeghi, A., and Shastry, B. Towards taming privilege-escalation attacks on android. In *Proc. of NDSS* (2012).

[17] Chen, K. Z., Johnson, N. M., D'Silva, V., Dai, S., MacNamara, K., Magrino, T. R., Wu, E. X., Rinard, M., and Song, D. X. Contextual policy enforcement in android applications with permission event graphs. In *NDSS* (2013).

[18] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 239–252.

[19] Christensen, A. S., Møller, A., and Schwartzbach, M. I. Precise analysis of string expressions. In *Proceedings of the 10th international conference on Static analysis (SAS'03)* (2003), pp. 1–18.

[20] Corp., S. 2012 norton study: Consumer cybercrime estimated at $110 billion annually. http://www.symantec.com/about/news/release/article.jsp?prid=20120905_02, Sept. 2012.

[21] Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security (ISC)* (2010).

[22] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. Quire: Lightweight provenance for smart phone operating systems. In *Proc. of USENIX* (2011).

[23] Enck, W., Gilbert, P., Chun, B. g., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of USENIX OSDI* (2011).

[24] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2009).

[25] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 627–638.

[26] Felt, A. P., Wang, H., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *Proc. of the 20th USENIX Security Symposium* (2011).

[27] Fragkaki, E., Bauer, L., Jia, L., and Swasey, D. Modeling and enhancing android's permission system. In *Proc. of ESORICS* (2012).

[28] Fuchs, A. P., Chaudhuri, A., and Foster, J. S. Scandroid: Automated security certification of android applications, 2009.

[29] Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing* (2012), Springer, pp. 291–307.

[30] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security* (2012).

[31] Gulwani, S. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming* (New York, NY, USA, 2010), PPDP '10, ACM, pp. 13–24.

[32] HOLAVANALLI, S., MANUEL, D., NANJUNDASWAMY, V., ROSENBERG, B., SHEN, F., KO, S. Y., AND ZIAREK, L. Flow permissions for android. In *Proceeding of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013).

[33] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 639–652.

[34] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 639–652.

[35] INC., G. Gartner reveals top predictions for IT organizations and users for 2012 and beyond. `http://www.gartner.com/newsroom/id/1862714`, Dec. 2011.

[36] JACKSON, D. Alloy: a lightweight object modelling notation. *TOSEM 11*, 2 (2002), 256–290.

[37] KLIEBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (2014), pp. 1–6.

[38] LI, L., BARTEL, A., KLEIN, J., TRAON, Y. L., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. Tech. Rep. arXiv preprint arXiv:1404.7431, 2014.

[39] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012).

[40] MAHMOOD, R., MIRZAEI, N., AND MALEK, S. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China, November 2014), FSE '14, ACM.

[41] MANN, C., AND STAROSTIN, A. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2012), SAC'12, ACM, pp. 1457–1462.

[42] NELSON, T., SAGHAFI, S., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. Aluminum: Principled scenario exploration through minimality. In *Proceedings of the International Conference on Software Engineering* (2013), pp. 232–241.

[43] OCTEAU, D., MCDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND TRAON, Y. L. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Security Symposium* (Washington, DC, August 2013).

[44] RASTHOFER, S., ARZT, S., AND BODDEN, E. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security (NDSS 2014)* (2014).

[45] RASTHOFER, S., ARZT, S., LOVAT, E., AND BODDEN, E. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on* (2014), IEEE, pp. 40–49.

[46] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. *Computer 29*, 2 (1996), 38–47.

[47] SCHRECKLING, D., POSEGGA, J., KÖSTLER, J., AND SCHAFF, M. Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *Proceedings of the 6th IFIP WG 11.2 International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems* (Berlin, Heidelberg, 2012), WISTP'12, Springer-Verlag, pp. 208–223.

[48] SHABTAI, A., FLEDEL, Y., KANONOV, U., ELOVICI, Y., DOLEV, S., AND GLEZER, C. Google android: A comprehensive security assessment. *Security & Privacy, IEEE 8*, 2 (2010), 35–44.

[49] SRIVASTAVA, S., GULWANI, S., AND FOSTER, J. S. From program verification to program synthesis. In *POPL'10* (Jan. 2010), pp. 313–326.

[50] SRIVASTAVA, S., GULWANI, S., AND FOSTER, J. S. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* (Jan. 2012).

[51] TORLAK, E., AND BODIK, R. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming &#38; Software* (New York, NY, USA, 2013), Onward! '13, ACM, pp. 135–152.

[52] TRIPP, O., PISTOIA, M., COUSOT, P., COUSOT, R., AND GUARNIERI, S. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 210–225.

[53] VALLE É-RAI, R., CO, P., GAGNON, E., HENDREN, L., AND LAM, P.AND SUNDARESAN, V. Soot - a java bytecode optimization framework. In *Proc. of CASCON'99* (1999).

[54] WANG, X., SUN, K., WANG, Y., AND JING, J. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. of 18th Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[55] WEI, F., ROY, S., OU, X., AND ROBBY. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)* (2014).

[56] WU, L., GRACE, M., ZHOU, Y., WU, C., AND JIANG, X. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013).

[57] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium* (2012), pp. 539–552.

[58] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 95–109.

[59] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS 2013)* (2013).

[60] ZHOU, Y., Y. WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)* (2012).