Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android

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 SEPAR, combines static 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, SEPAR has proven to be highly effective at detecting previously unknown vulnerabilities as well as preventing their exploitation.**

I. 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 [52], 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 [30] 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 iOS, as 38% of mobile users have experienced cybercrime [52]. This is, though, nowhere more evident than in the Android market, where many cases of apps infected with malware and spyware have been reported [49].

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 [24], [31], [37], [42], analysis of the least-privilege principle [25], [27], and enhancements to Android protection mechanisms [18], [23], [28]. 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 [38], [39], [56] and permission leaks [16], [36], 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 malware are embedded in supposedly normal apps that aim to leverage vulnerabilities in either the platform or other apps on the market for nefarious purposes [51]. 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 for **s**ynthesis and **e**nforcement of security **p**olicies for **A**nd**r**oid, called SEPAR. It combines scalable static analysis with lightweight formal methods. SEPAR 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.

SEPAR 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

[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.

```java
1  public class LocationFinder extends Service {
2   public void onStartCommand(Intent intent, int flags, int
         startId){
3   LocationManager lm = getSystemService(Context.
        LOCATION_SERVICE);
4   Location lastKnownLocation =
5   lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
6   Intent intent = new Intent();
7   intent.setAction("showLoc");
8   intent.putExtra("locationInfo", lastKnownLocation.
        toString());
9   startService(intent);
10  ... }
```

Listing 1: LocationFinder sends the retrieved location data to another component of the same app via implicit Intent messaging.

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 SEPAR framework*: We describe SEPAR—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. SEPAR is publicly available for download [10].
- *Experiments*: We present results from experiments run on 4,000 real-world apps as well as DroidBench2.0 test suite [4], corroborating SEPAR'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 II motivates our research through an illustrative example. Section III provides an overview of SEPAR. Sections IV, V and VI describe the details of static model extraction, formal synthesis and dynamic enforcement of policies, respectively. Sections VII and VIII present implementation and evaluation of the research. The paper concludes with an outline of the related research and future work.

II. Motivating Example

To motivate the research and illustrate our approach, we provide an example of a vulnerability pattern having to do with inter-component communication (ICC) among Android apps. Android provides a flexible model of component communication using a type of application-level message known as *Intent*. A typical app is comprised of multiple components (e.g., Activity, Service) that communicate using Intent messages. In addition, under certain circumstances, an app's component could send Intent messages to another app's components to perform actions (e.g., take picture, send text message, etc.). Figure 1 partially shows a bundle of two benign, yet vulnerable apps, installed together on a device.

The first application is a navigation app that obtains the device location (GPS data) in one of its components and sends

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

Listing 2: MessageSender receives an Intent and sends a text message.

it to another component of the app via Intra-app Intent messaging. The Intent involving the location data (Listing 1, lines 3–9), instead of explicitly specifying the receiver component, i.e., *RouteFinder* service, implicitly specifies it through declaring a certain `action` to be performed in that component. This represents a common practice among developers, yet an anti-pattern that may lead to unauthorized Intent receipt [21], 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 vulnerability of the second application, a messenger app, occurs on line 11 of Listing 2, where *MessageSender*, specified as a public component in the app manifest file, 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 MessageSender 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.

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. 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 not only identifies the vulnerabilities and capabilities in

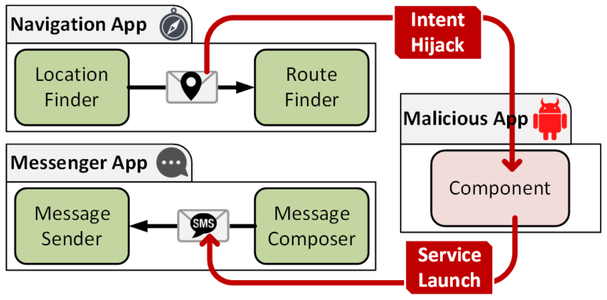


Fig. 1: A potential malicious application—its signature automatically generated by SEPAR—leverages vulnerabilities in other already installed benign applications to perform actions (like sending device location through text messages) that are beyond its individual privileges. As the Android access control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

individual apps, but also 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 SEPAR and then delve into more details about its approach to address these issues.

III. 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, SEPAR 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 uses lightweight formal analysis techniques [3] to find vulnerabilities in the extracted app models, and generates 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 dalvik 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

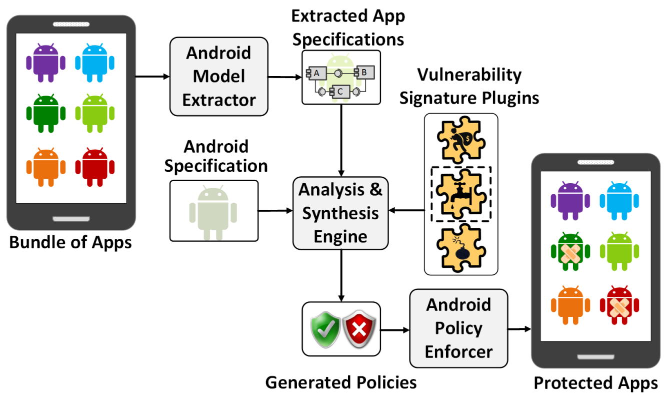


Fig. 2: Approach Overview.

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 [3]. Alloy is a formal specification language based on relational logic, amenable to fully automated yet bounded analysis.

SEPAR 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 SEPAR prototype supports inter-component vulnerabilities, such as Activity/Service launch, Intent hijack, privilege escalation, and information leakage [18], [21], [33]. Its plugin-based architecture supports the necessary extensions that can be provided by users 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 [3] and Aluminum [43], 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, SEPAR 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 monitors each vulnerable app at runtime 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, SEPAR 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.

IV. AME: Android Model Extractor

The AME module, that individually analyzes each app to extract a model of its behavior, is built upon state-of-the-art static analysis techniques for the Android framework. This section describes the extraction process, with an emphasis

Algorithm 1: Update Passive Intent Target

```
Input: Intents: Set of all identified Intents
Output: Target components for passive Intents
for p in Intents do
    if p.isPassiveIntent then
        for i in Intents do
            if i.hasRequestResult & i.target = p.sender then
                p.addTarget(i.sender)
            end
        end
    end
end
```

on the important improvements on prior work. Due to space limitations, we have made the detailed algorithms and implementation of our extensions available at [10].

Architecture Extraction. To obtain an app model, AME first examines the app manifest file to capture the high-level architectural information, including the components comprising the app, permissions that the app requires, and the enforced permissions 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 of components.

Intent Extraction. The next step of model extraction involves an inter-procedural data flow analysis [17], to track the Intents and Intent Filters that are declared in code, rather than the manifest file, as well as their properties. Each Intent belongs to one particular component that sends it, may have one recipient component and may include an action, data and a set of categories. The action field specifies the general action to be performed in the recipient component; the data field represents additional information about the data to be processed by the action; and the categories filed specifies the kind of component that should handle the Intent. An Intent can also include extra data. Similar to Intents, each Intent Filter has a non-empty set of actions and two sets of data and categories. 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 [22], 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* [53]. 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.

There are some special cases in implicit invocations of inter-component entry points, where the caller method triggers a two-way communication between components. Examples include `bindService` and `startActivityForResult`. A component, for instance, can use `startActivityForResult` to start another component, which itself implicitly calls the first component with a new Intent embodying the results once finishes running. However, the returning implicit Intent, which we call *passive Intent*, includes no information (e.g., action and category) specifying its target component, making it difficult for static analyzers to identify the receiver in this second implicit invocation. Algorithm 1 outlines identifying target components for passive Intents. The logic of the algorithm is as follows. For each passive Intent, p, look up Intents that both request for results and their target components match senders of p. Insert the senders of such Intents into the target set of p.

Path Extraction. AME analyzes the app using a static taint analysis to track sensitive data flow tuples $< Source, Sink >$, 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. To achieve a high precision in data flow analysis, our approach is flow-, field-, and context-sensitive [13], 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, SEPAR's analysis, however, is not path-sensitive. The results (cf. Sec. VIII) 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 [45], identified through the use of machine-learning techniques. To further detect those paths traversing through different components, we adapted this set by identifying source and sink methods corresponding to inter-component communication. The identified sensitive data flows 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 [14], 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.

A node could be directly tagged as *permission-required* node, 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 exposed permissions of that component.

V. ASE: Analysis and 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 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 3, 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 optimized for automated analysis, with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [3].

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.

Alloy is an appropriate language for our modeling and synthesis purposes for several reasons: (1) its simple set theoretic language, backed with logical and relational operators, was sufficiently expressive for formal declarative specification of both applications and properties to be checked; (2) its ability to automatically analyze specifications is useful as an automation mechanism, enabling automatic synthesis of violation scenarios as satisfying solutions; finally, (3) the formal analyzers available for Alloy (e.g., [43]) translate our high-level model specifications into a SAT formula that can be solved by off-the-shelf SAT solvers, and thereby enable utilizing state-of-

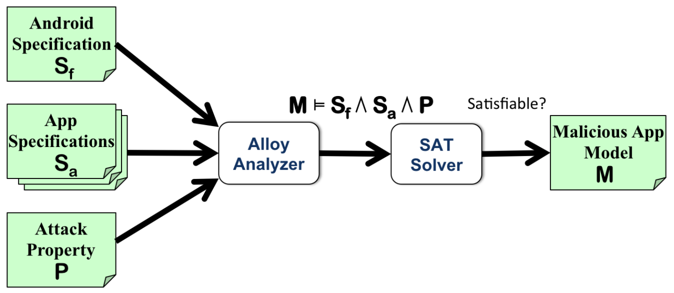


Fig. 3: Automated synthesis of possible exploit specifications.

the-art constraint solvers for our model synthesis. The rest of this section first provides a brief overview of Alloy, and then details different parts of implementing the synthesis problem.

Alloy Overview. Alloy is a declarative language based on the first-order relational logic with transitive closure [3]. The inclusion of transitive closure extends its expressiveness beyond first-order logic. Essential data types, that collectively define the vocabulary of a system, are specified in Alloy by their type signatures (sig). Signatures represent basic types of elements, and the relationships between them are captured by the the declarations of *field*s within the definition of each signature. Consider the following Alloy model. It defines two Alloy signatures: `Application` and `Component`. The `cmps` relation is defined over these two signatures.

```
sig Application{
  cmps: Component
}
sig Component{}
```

Analysis of specifications written in Alloy is completely automated, based on transformation of Alloy's relational logic into a satisfiability problem. Off-the-shelf SAT solvers are then used to exhaustively search for either satisfying models or counterexamples to assertions. To make the state space finite, certain scopes need to be specified that limit the number of instances of each type signature. The following specification asks for instances that contain at least one `Component`, and specifies a scope that bounds the search for instances with at most two objects for each top-level type (`Application` and `Component` in this example).

```
pred modelInstance{ some Component }
run modelInstance for 2
```

When executed, the Alloy Analyzer produces model instances, two of which are shown in Fig. 4. The model instance of Fig. 4a includes one application and two components, one of them belongs to no application. Fig. 4b shows another model instance with two applications, each one having one component.

Facts (fact) are formulas that take no arguments, and define constraints that every instance of a model must satisfy, thus restricting the instance space of the model. The following fact paragraph, for example, states that each `Component` should belong to exactly one `Application`. Re-executing the Alloy Analyzer produces a new set of model instances, where while Fig. 4b is still a valid instance, model of Fig. 4a is eliminated.

```
fact {
   all c: Component| one c.~cmps
}
```

The other essential constructs of the Alloy language include: *Predicates*, *Functions* and *Assertions*. Predicates (pred) are named logical formulas used in defining parameterized and reusable constraints that are always evaluated to be either true or false. Functions (fun) are parameterized expressions. A function similar to a predicate can be invoked by instantiating its parameter, but what it returns is either a true/false or a relational value instead. An assertion (assert) is a formula required to be proved. It can be used to check a certain property of a model.

The Alloy language comes with a set of logical and relational operators. The dot (.) and tilde ($\sim$) operators denote

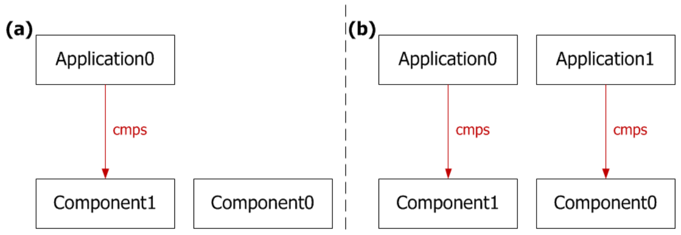


Fig. 4: Two model instances of the above Alloy specification.

a relational join of two relations and the transpose operation over a binary relation, respectively. The transitive closure (ˆ) of a relation is the smallest enclosing relation that is transitive. The reflexive-transitive closure (*) of a relation is the smallest enclosing relation that is both transitive and reflexive.

We will introduce additional details of the Alloy language as necessary to present our policy synthesis approach. For further information about Alloy, we refer the interested reader to [3].

Formal Model of Android Framework. Listing 3 shows (part of) the Alloy code describing the meta-model for Android application models. The complete version of all Alloy models that appear in this paper are available at [10]. Our model is based on the official Android documentation [32]. Android is a large and complex operating system, and modeling it in its entirety would be infeasible. Thus, we focused on the parts of Android that are relevant to the inter-component communication and their potential security challenges. For example, note the signatures `Component` and `Intent`. Signatures defined as abstract represent types of elements that cannot have an instance object without explicitly extending them. 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 in the context of this component. We define the source and destination of a path based on canonical permission-required resources identified by Holavanalli et al. for Android applications [36]. 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 ICC mechanism augments both source and destination sets. Note that to eliminate private components from inter-app analysis, SEPAR considers the component's exported attribute. In fact, a component can receive Intents from other applications, or is public, if its exported attribute is set or contains at least one Intent filter. Such elimination of private components from inter-app analysis also contributes to the scalability of the approach (i.e., less components to be analyzed).

```
abstract sig Component{
  app: one Application,
  intentFilters: set IntentFilter,
  permissions: set Permission,
  paths: set DetailedPath
}
abstract sig IntentFilter{
  actions: some Action,
  dataType: set DataType,
  dataScheme: set DataScheme,
  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,
  receiver: lone Component,
  action: lone Action,
  categories: set Category,
  dataType: lone DataType,
  dataScheme: lone DataScheme,
  extra: set Resource
}
```

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

```
(a) App1 model
open androidDeclaration
...
one sig LocationFinder extends Service{}{
  app in App1
  no intentFilters
  paths = pathLocationFinder1
  permissions = ACCESS_FINE_LOCATION
}
one sig pathLocationFinder1 extends Path{}{
  source = LOCATION
  sink = ICC
}
one sig Intent1 extends Intent{}{
  sender = LocationFinder
  no receiver
  action=showLoc
  categories= DEFAULT
  no dataType
  no dataScheme
  extra= LOCATION
}
(b) App2 model
one sig MessageSender extends Service{}{
  app in App2
  intentFilter = IntentFilter1
  paths = pathMessageSender1
  no permissions
}
one sig pathMessageSender1 extends Path{}{
  source = ICC
  sink = SMS
}
```

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

The fact `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 may also include an `action`, `data` (`type` and `scheme`) and a set of `categories`.[2] These 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

[2]The multiplicity keyword ***some*** in Alloy denotes that the declared `IntentFilter.actions` relation contains at least one element; the keyword ***set*** tells Alloy that `categories` map each `IntentFilter` object to zero or more `Category` objects, and the keyword ***lone*** indicates that this `Intent.component` is optional, and an Intent may have one or no declared recipient component.

```
1  sig GeneratedServiceLaunch{
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 = ICC
10   some malIntent.extra
11   malCmp in Activity
12 }
```

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

it cannot contain less. The `extra` field indicates the types of resources carried by the Intent.

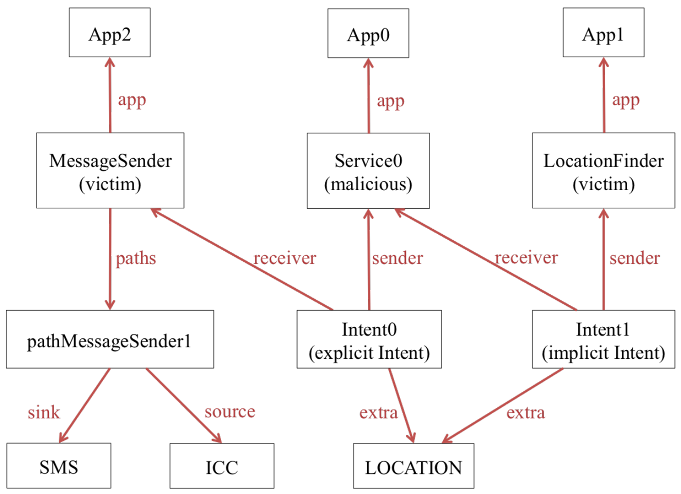
Formal Model of Apps. Listing 4 partially shows the Alloy specifications for the apps shown in Listings 1 and 2. As already mentioned (cf. Section IV), these app specifications are automatically extracted by the AME component from each Android application. Each app specification starts by importing the *androidDeclaration* module (cf. Listing 3). Among other things, the `LocationFinder` component contains a sensitive path (`pathLocationFinder1`), that represents a data-flow from where the sensitive GPS data is retrieved, to an Intent event message. The `extra` field of the Intent in the generated Alloy model (line 21) is accordingly set. The `path` field of the *MessageSender* in the generated Alloy model (lines 27, 30–33) reflects another data-flow path, started from an IntentFilter and reaches to a node, which uses the data in the body of a text message. Note that this component does not enforce any access permission neither in the manifest file nor in the code (line 28).

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. Specifically, each vulnerability model captures a specific type of inter-component communication security threat, according to those identified by Chin et al. [21] and Bugiel et al. [18]. The security property check is then formulated as a problem of finding a valid trace that satisfies the vulnerability signature specifications. If the Alloy Analyzer finds a solution to this problem, the property is violated; the returned solution encodes an exact scenario (states of all elements, such as components and Intents) leading to the violation. 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 `GeneratedServiceLaunch` signature along with its *signature fact* that specifies the elements involved in, and the semantics of, a service launch exploit, respectively. In short, a malicious component (`malCmp`) can launch a component 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, SEPAR 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, among others:



The diagram is accurate for the result that the analyzer computed, but we have edited it to omit some details for readability. It essentially states the scenario represented in Figure 1, in which a postulated malicious component, here the generated `App0/Service0` component, can send the device location data captured from a vulnerable Intent, `Intent1` (cf. Listing 4, lines 14–22), to the desirable phone number via an explicit Intent, `Service0/Intent0`, sent to the `App2/MessageSender` component that is vulnerable to service launch. Here the analysis has found that it is possible to devise a malicious capability that can leverage the vulnerabilities in the apps installed on the device for nefarious purposes. Given this, SEPAR formulates a policy, as described next, that prevents certain Intent-based interactions from occurring to prevent the exploitation of vulnerabilities, thereby achieving proactive defense if such a malicious capability were to be installed on the device.

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

VI. APE: Android Policy Enforcer

In the implementation of APE, we faced three possible alternatives: (1) modify the Android OS to enforce the policies, (2) modify an app through injection of policy enforcement logic into the app's implementation by instrumenting the APK file, and (3) dynamic memory instrumentation of the app's process. We chose the third approach, as it allows SEPAR 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 [47], our approach is comprised of two elements: *policy decision 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 ICC calls.

Our Android policy enforcer relies on the Xposed [11] framework for modifying the behavior of Android apps at runtime, without making any changes in the apps' APK files. It provides mechanisms to "hook" method calls. A hook is a method that is called before or after a certain method, making it possible to control pre/post method call activities, by modifying a method's parameters, its return values, or even entirely skipping the call to the method.

The PDP is realized as an independent Android app that stores the synthesized policies for preventing or allowing ICC access. Our policies are in the form of event-condition-action (ECA) rules. The PEP in our case corresponds to an Xposed module to dynamically intercept event messages. More specifically, each ICC method in an app's APK file (e.g., `startService(Intent)`) is hooked, such that whenever it is invoked, it is first assessed to see whether the operation should proceed (e.g., `Intent` to be delivered to its destination) by calling the PDP. The major advantages of using runtime process instrumentation over modifying individual apps are scalability and framework generalization. Additionally, instrumentation of APK files changes the signature of apps, which might prevent their proper execution.

PEP hooks 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 ICC 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 ICC calls would naturally force the app to operate in a degraded mode.

Continuing with our running example, SEPAR generates the following policy, where the conditions in the generated ECA rule correspond to the properties of the malicious Intent in the synthesized vulnerability model instance.

```
{ event : ICC_received,
  condition : [{ Intent.extra: LOCATION},
               { Intent.receiver: MessageSender}],
  action : user_prompt
}
```

It states that every attempt of sending device LOCATION data through the MessageSender component 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 guarded with policies generated regarding Intent hijacking and Service Launch, respectively.

VII. Tool Implementation

We have implemented SEPAR as a publicly available tool [10]. We have built our static analysis capability on top of the Soot [54] framework. We used Flowdroid for intra-component taint analysis [13], and extended it to improve precision of analysis especially to support complicated ICC methods (cf. Section IV). The prototype implementation of SEPAR 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 is 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 [3]. As a back-end analysis engine, SEPAR relies on Aluminum [43], 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 Xposed framework [11] for preventing event messages violating synthesized policies.

VIII. Evaluation

This section presents the experimental evaluation of SEPAR. Our evaluation addresses the following research questions:

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

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

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

RQ4. What is the performance of SEPAR's policy enforcement?

A. *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 [4] and ICC-Bench [8] suites of benchmarks, two sets of Android applications containing ICC based privacy leaks for which all vulnerabilities are known in advance—establishing a ground truth. These test cases comprise the most frequently used ICC 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 SEPAR with existing tools targeted at ICC vulnerability detection, namely DidFail [38] and AmanDroid [56]. COVERT [16] 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

TABLE I: Comparison between SEPAR, 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	SEPAR
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	☐	☐	☐
Precision		55%	86%	100%
Recall		37%	48%	97%
F-measure		44%	63%	98%

that COVERT cannot detect. We also tried to run IccTA [39], 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.

Table I summarizes the results of our experiments for evaluating the accuracy of SEPAR in detecting ICC vulnerabilities compared to other state-of-the-art techniques. SEPAR 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 SEPAR'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 the vulnerabilities that are due to explicit Intents, such as information leak. The results show that SEPAR outperforms the other two tools in terms of both precision and recall.

B. Results for RQ2 (SEPAR and Real-World Apps)

To evaluate the implications of our tool in practice, we collected 4,000 apps from the following four different sources: **(1) 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. **(2) 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. **(3) Malgenome** [60]: 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. **(4) Bazaar** [2]: 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 [10]. Out of 4,000 apps, SEPAR 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 SEPAR'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

TABLE II: Experiments performance statistics.

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

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.

Privilege Escalation. *Ermete SMS* is a text messaging app with *WRITE_SMS* permission. Upon receiving an Intent, its *ComposeActivity* component 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.

C. *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 5 presents the time taken by SEPAR 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 II 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 SEPAR 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.

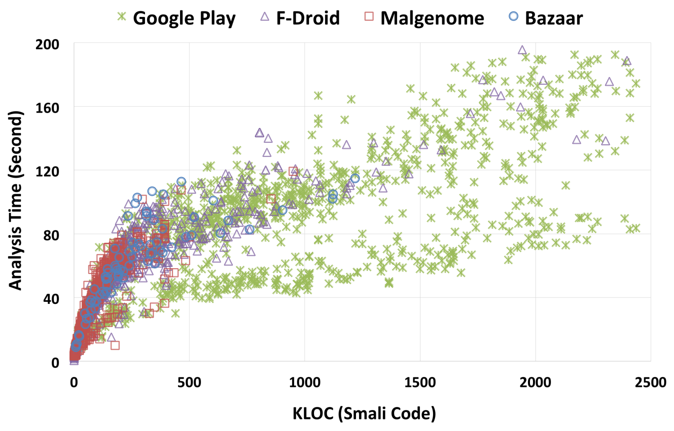


Fig. 5: Scatter plot representing analysis time for model extraction of Android apps.

Finally, we compared scalability of SEPAR with other tools that support analysis of inter-app vulnerabilities, namely DidFail and AmanDroid. Figure 6 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 SEPAR outperforms the two other tools in terms of scalability.

D. *Results for RQ4 (Policy Enforcement)*

The last evaluation criterion is the performance benchmark of SEPAR's policy enforcement. To measure the runtime overhead required for APE (i.e., policy enforcement), we have tested a set of benchmark applications. Our benchmark applications repeatedly perform several ICC operations, such as the *startService* method. We have handled uncontrollable factors in our experiments by repeating the experiments 33 times, the minimum number of repetitions needed to accurately measure the average execution time overhead at 95% confidence level. Overall, the execution time overhead incurred by APE for policy enforcement is 11.80% $\pm$ 1.76%, making the effect on user experience negligible. Note that using the run-time process instrumentation (cf. section VI), our infrastructure only introduces overhead with the ICC calls, and does not have any overhead in terms of the non-ICC calls. Thus, in practice, the overhead introduced by our approach is significantly less than 11.80%.

IX. 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.

Static analysis. A large body of work [19], [21], [24], [29], [31], [33], [44], [58] focuses on performing program analysis over Android applications for security. Chin et al. [21] studied security challenges of Android communication, and developed ComDroid to detect those vulnerabilities through static analysis of each app. Octeau et al. [44] developed

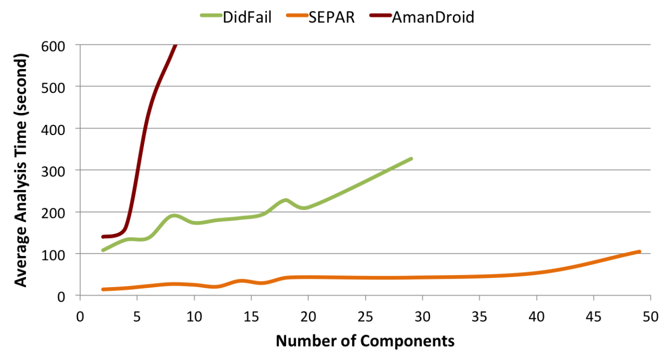


Fig. 6: Performance Comparison Results. To improve the readability, the y-axis is limited to 10 minutes (600 seconds), which intercepts Amandoird line at 8-components point. DidFail was unable to analyze apps with more than 30 components.

Epicc for analysis of Intent properties—except data scheme—through inter-procedural data flow analysis. FlowDroid [13] introduces a precise approach for static taint flow analysis in the context of each application component. CHEX [40] 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. SEPAR'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 [16], [38], [39], [56], [58]. DidFail [38] 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 SEPAR. IccTA, similarly, leverages an intent resolution analysis to identify inter-component privacy leaks [39]. 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 [56]. 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 [16] 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.

Policy enforcement. The other relevant thrust of research has focused on policy enforcement [15], [20], [35], [37], [46], [48], [55], [57]. Kirin [25] 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, SEPAR is more precise as it dynamically analyzes policy violations against flows that actually occur at run-time.

Along the same line, some other techniques enforce policies at runtime. Among others, Kynoid [48] performs a dynamic taint analysis over a modified version of Dalvik VM. This approach, similar to many of the previously proposed solutions [23]–[26], requires changes to the Android. ASM [35] presents an extensible security modules framework that enables apps to define hooks in order to enforce app-specific security requirements. While this work is concerned with the design and implementation of a programmable interface for defining new reference monitors, it does not consider the problem that we address, the automation of synthesizing ICC policies. The two approaches are thus complementary in that SEPAR's APE module can be realized as ASM hooks. More recently, DeepDroid [55] presents a policy enforcement scheme based on dynamic memory instrumentation of system processes. However, it depends on undocumented internal architecture of Android framework and its system resources which may change in future versions without notice.

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

Synthesis. Finally, constraint solving for synthesis and analysis has increasingly been used in a variety of domains [12], [34], [50]. 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, SEPAR 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, SEPAR is the first formally-precise technique for automated synthesis and dynamic enforcement of Android security policies.

X. Concluding Remarks

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 SEPAR, 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-grain 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 SEPAR's ability in finding previously unknown vulnerable apps as well as preventing their exploitation.

The great majority of Android devices run KitKat or older versions [1], which provide a static permission model. However, a recently released version of Android (Marshmallow) provides a Permission Manager that allows users to revoke granted permissions after installation time. We believe a solution such as SEPAR becomes even more relevant in this new version of Android, where the policies have to be fine-tuned to the user-specific, continuously-evolving configuration of apps. SEPAR has more potential in such a dynamic setting, as it can be applied to continuously verify the security properties of an evolving system as the status of app permissions changes. SEPAR's incremental analysis for policy synthesis can then be performed on permission-modified apps at runtime. In cases where vulnerabilities are detected and new policies are synthesized, mitigation strategies could be carried through the policy enforcer deployed on mobile devices, restricting communications between certain apps to secure the system.

Our approach has a few limitations. Current implementation of SEPAR mainly monitors 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 [59]. 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 [24] and EvoDroid [41], that would allow us to extract additional behaviors that might be latent in apps.

REFERENCES

[1] "Android platform versions," http://developer.android.com/about/dashboards/index.html#2015.

[2] "Bazaar," http://cafebazaar.ir/.

[3] *D. Jackson, Software Abstractions, 2nd ed. MIT Press, 2012.*

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

[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] "Separ," http://doiop.com/separ.

[11] "Xposed framework," http://repo.xposed.info/.

[12] D. Akhawe, A. Barth, P. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *Proc. of CSF*, 2010.

[13] 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 PLDI*, 2014.

[14] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proc. of CCS*, 2012.

[15] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard–enforcing user requirements on android apps," in *Proc. of TACAS*, 2013.

[16] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, 2015.

[17] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proce. of SOAP*, 2012.

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

[19] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *Proc. of NDSS*, 2015.

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

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

[22] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. of SAS*, 2003.

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

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

[25] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. of CCS*, 2009.

[26] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. of USENIX Security*, 2011.

[27] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. of CCS*, 2011, pp. 627–638.

[28] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing android's permission system," in *Proc. of ESORICS*, 2012.

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

[30] Gartner Inc., "Gartner reveals top predictions for IT organizations and users for 2012 and beyond," http://www.gartner.com/newsroom/id/1862714, 2011.

[31] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Trust and Trustworthy Computing*, 2012.

[32] Google, "Android api reference document," http://developer.android.com/reference.

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

[34] S. Gulwani, "Dimensions in program synthesis," in *Proc. of PPDP*, 2010.

[35] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "Asm: A programmable interface for extending android security," in *Proc. of USENIX*, 2014.

[36] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek, "Flow permissions for android," in *Proc. of ASE*, 2013.

[37] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proc. of CCS*, 2011, pp. 639–652.

[38] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proc. of SOAP*, 2014.

[39] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proc. of ICSE*, 2015.

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

[41] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proc. of FSE*, 2014.

[42] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *Proc. of SAC*, 2012.

[43] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Aluminum: Principled scenario exploration through minimality," in *Proc. of ICSE*, 2013, pp. 232–241.

[44] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," in *Proc. of USENIX Security*, 2013.

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

[46] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "Droidforce: Enforcing complex, data-centric, system-wide policies in android," in *Proc. of ARES*, 2014, pp. 40–49.

[47] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[48] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff, "Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android," in *Proc. of WISTP*, 2012, pp. 208–223.

[49] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *Security & Privacy, IEEE*, vol. 8, no. 2, pp. 35–44, 2010.

[50] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *POPL'10*, Jan. 2010, pp. 313–326.

[51] Symantec, "2015 internet security threat report," Tech. Rep. Vol. 20, Apr. 2015.

[52] Symantec Corp., "2012 norton study." http://www.symantec.com/about/news/release/article.jsp?prid=20120905_02, Sep. 2012.

[53] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Proc. of FASE*, 2013.

[54] R. Valle é-Rai, P. Co, E. Gagnon, L. Hendren, and V. Lam, P.and Sundaresan, "Soot - a java bytecode optimization framework," in *Proc. of CASCON'99*, 1999.

[55] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices," in *Proc. of NDSS*, 2015.

[56] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proc. of CCS*, 2014.

[57] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications." in *Proc. of USENIX Security*, 2012.

[58] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proc. of NDSS*, 2013.

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

[60] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. of IEEE Security and Privacy*, 2012.