This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2419611, IEEE Transactions on Software Engineering

COVERT: Compositional Analysis of Android Inter-App Permission Leakage

Hamid Bagheri, Alireza Sadeghi, Joshua Garcia and Sam Malek, *Member, IEEE*

Abstract—Android is the most popular platform for mobile devices. It facilitates sharing of data and services among applications using a rich inter-app communication system. While access to resources can be controlled by the Android permission system, enforcing permissions is not sufficient to prevent security violations, as permissions may be mismanaged, intentionally or unintentionally. Android's enforcement of the permissions is at the level of individual apps, allowing multiple malicious apps to collude and combine their permissions or to trick vulnerable apps to perform actions on their behalf that are beyond their individual privileges. In this paper, we present *COVERT*, a tool for compositional analysis of Android inter-app vulnerabilities. COVERT's analysis is modular to enable incremental analysis of applications as they are installed, updated, and removed. It statically analyzes the reverse engineered source code of each individual app, and extracts relevant security specifications in a format suitable for formal verification. Given a collection of specifications extracted in this way, a formal analysis engine (e.g., model checker) is then used to verify whether it is safe for a combination of applications—holding certain permissions and potentially interacting with each other—to be installed together. Our experience with using COVERT to examine over 500 real-world apps corroborates its ability to find inter-app vulnerabilities in bundles of some of the most popular apps on the market.

Index Terms—Formal Verification, Static Analysis, Android, Inter-App Vulnerabilities

1 Introduction

Mobile app markets are creating a fundamental paradigm shift in the way software is delivered to the end users. The benefits of this software supply model are plenty, including the ability to rapidly and effectively acquire, introduce, maintain, and enhance software used by the consumers. By providing a medium for reaching a large consumer market at a nominal cost, app markets have leveled the software development industry, allowing small entrepreneurs to compete with prominent software development companies. Application frameworks are the key enablers of these markets. An application framework, such as the one provided by Android, ensures apps developed by a wide variety of suppliers can interoperate and coexist together in a single system (e.g., a phone) as long as they conform to the rules and constraints imposed by the framework.

This paradigm shift, however, has given rise to a new set of security challenges. In parallel with the emergence of app markets, we are witnessing an increase in the security threats targeted at mobile platforms. This is nowhere more evident than in the Android market (i.e., Google Play), where many cases of apps infected with malwares and spywares have been reported [1]. Numerous culprits are at play here, and some are not even technical, such as the general lack of an overseeing authority in the case of open markets and inconsequential implication for those caught provisioning applications with vulnerabilities or malicious capabilities.

In this context, Android's security has been a thriving subject of research in the past few years. Leveraging program analysis techniques, these research efforts have investigated weaknesses from various perspectives, including detection of information leaks [2]–[4], analysis of the least-privilege principle [5], [6], and enhancements to Android protection mechanisms [7]–[9]. The majority of these approaches, however, are subject to a common limitation: they are intended to detect and mitigate vulnerabilities in a single app, but fail to identify vulnerabilities that arise due to the interaction of multiple apps. Vulnerabilities due to the interaction of multiple apps, such as collusion attacks and privilege escalation chaining [5], cannot be detected by techniques that analyze a single app in isolation. Thus, security analysis techniques in such domains need to become compositional in nature.

This paper contributes a novel approach, called COVERT, for compositional analysis of Android inter-app permission leakage vulnerabilities. Unlike all prior techniques that focus on assessing the security of an individual app in isolation, our approach has the potential to greatly increase the scope of application analysis by inferring the security properties from individual apps and checking them as a whole by means of formal analysis. This, in turn, enables reasoning

- *H. Bagheri is with both the department of Computer Science at George Mason University and the Computer Science and Artificial Intelligence Laboratory at MIT. E-mail: hbagheri@mit.edu*
- *A. Sadeghi, J. Garcia and S. Malek are with the Department of Computer Science, George Mason University, Fairfax, VA, 22030. E-mail: {asadeghi,jgarci40,smalek}@gmu.edu*

about the overall security posture of a system (e.g., a phone device) in terms of the security properties inferred from the individual apps.

COVERT combines static analysis with formal methods. At the heart of our approach is a modular static analysis technique for Android apps, designed to enable incremental and automated checking of apps as they are installed, removed, or updated on an Android device. Through static analysis of each app, our approach extracts essential information and captures them in an analyzable formal specification language. These formal specifications are intentionally at the architectural level to ensure the technique remains scalable, yet represent the true behavior of the implemented software, as they are automatically extracted from the installation artifacts. The set of models extracted in this way are then checked as a whole for vulnerabilities that occur due to the interaction of apps comprising a system. COVERT uses *Alloy* as a specification language [10], and the *Alloy Analyzer* as the analysis engine. Alloy is a formal specification language based on first order logic, optimized for automated analysis.

Since COVERT's analysis is compositional, it provides the analysts with information that is significantly more useful than what is provided by prior techniques. Our experiences with a prototype implementation of the approach and its evaluation against one of the most prominent inter-app vulnerabilities, i.e. privilege escalation, in the context of hundreds of real-world Android apps collected from variety of repositories have been very positive. The results, among other things, corroborate its ability to find vulnerabilities in bundles of some of the most popular apps on the market.

Contributions. This paper makes the following contributions:

- *Formal model of Android framework:* We develop a formal specification representing the behavior of Android apps that is relevant for the detection of inter-app permission leakage vulnerabilities. We construct this formal specification as a reusable Alloy module to which all extracted app models conform.
- *Modular analysis:* We show how to exploit the power of our formal abstractions by building a modular model extractor that uses static analysis techniques to automatically extract formal specifications (models) of apps form their installation artifacts.
- *Implementation:* We develop a prototype implementation on top of our formal framework for compositional security analysis of Android apps.
- *Experiments*: We present results from experiments run on over 500 real-world apps, corroborating COVERT's ability in effective compositional analysis of Android inter-app permission leakage vulnerabilities in the order of minutes.

Outline. The remainder of this paper is organized as follows. Section 2 provides the background knowledge required to understand the contributions of our work. Section 3 motivates our research through an illustrative example. Section 4 provides an overview of COVERT. Sections 5 and 6 describe the details of model extraction and formal analysis, respectively. Section 7 presents the evaluation of the research. Finally, the paper concludes with a discussion of limitations, and an outline of the related research and future work.

2 Android Overview

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

Application Components. Components are basic logical building blocks of Android applications. Each component can be run individually, either by its embodying application or by system upon permitted requests from other applications. Android applications can comprise 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. Playing 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-app data persistence as well as sharing data across applications.

Inter-Process Communication. As part of its protection mechanism, Android insulates applications from each other and system resources from applications via a sandboxing mechanism. Such application insulation that Android depends on to protect applications requires interactions to occur through a message passing mechanism, called inter-process communication (IPC). IPC is 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 invocations come in different flavors, e.g., explicit or implicit, intra- or inter-apps, etc. Android's IPC 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.

Application Configuration. Each Android application must declare upfront its configuration. Among other things, it describes the principal components that constitute the application, along with their types and capabilities. Component capabilities are specified as a set of *Intent Filters* that represent the kinds of requests a given component can respond to. Such high-level application descriptions are documented in a separate XML file, called *manifest*, that accompanies the application.

Permissions. Enforcing permissions is the other mechanism, besides sandboxing, provided by the Android framework to protect applications, by which restrictions are placed on the specific operations that an application can perform, such as interacting with the system APIs and databases, as well as cross-application interactions. Each application must declare upfront as part of its manifest the permissions it requires, and the Android system prompts the user for consent during the application installation. Should the user refuse granting the requested permissions to an application, the application installation is canceled. No dynamic mechanism is provided by Android for granting permissions after application installation. The manifest file also declares permissions enforced by the application or by any of its components; the other applications thus must have those permissions in order to interact with such protected components. Android platform provides over 130 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 permission mechanism has proved insufficient to prevent security violations, since permissions may be misused, intentionally or unintentionally, as illustrated in the next section.

3 Motivating Example

To motivate the research and illustrate our approach, we provide an example of a vulnerability pattern having to do with Inter-Process Communication (IPC) among Android apps. Android provides a flexible model of IPC using a type of application-level message known as *Intent* (cf. Section 2). A typical app is comprised of multiple processes (e.g., Activity, Service) that communicate using Intent messages. In addition, under certain circumstances, an app's processes could send Intent messages to another app's processes to perform actions (e.g., take picture, send text message, etc.). As an example, Listing 1 shows *CallerActivity* belonging to a malicious app sending an Intent message to *PhoneActivity* (Listing 2) belonging to a vulnerable app for placing a call to a premium-rate telephone number.

```
public class CallerActivity extends Activity {
  public void onCreate (Bundle savedInstanceState) {
  ...
  String action;
  if(selectedMenu == 1)
    action = "PHONE_CALL";
  else
    action = "PHONE_TEXT_MSG";
  btnOK = (Button) findViewById(R.id.btnOK);
  btnOK.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
    Intent intent = new Intent (action);
    intent.setClassName ("com.phoneservice", "com.
        phoneservice.PhoneActivity");
    intent.putExtra ("PHONE_NUM", "900−512−1677");
    startActivity (intent);
  }
  }
}
```

Listing 1: Malicious app: sends an Intent to call a premium-rate phone number.

```
1  public class MainActivity extends Activity  {
2    public void onCreate(Bundle savedInstanceState)  {
3    ...
4    Intent intent = new Intent (this, PhoneActivity.
         class);
5    startActivity (intent);
6    }
7  }
8
9  public class PhoneActivity extends Activity  {
10
11   public void onCreate(Bundle savedInstanceState)  {
12    ...
13    Intent intent = getIntent();
14    String number = intent.getStringExtra("PHONE_NUM");
15    //if (hasPermission())
16       makePhoneCall(number);
17    else
18    ...
19   }
20
21   boolean hasPermission ()  {
22     if(checkCallingPermission("android.permission.
           CALL_PHONE")==PackageManager.
           PERMISSION_GRANTED)
23     return true;
24   return false;
25   }
26
27   void makePhoneCall (String number) {
28   Intent callIntent = new Intent (Intent.ACTION_CALL);
29   callIntent.setData (Uri.parse(number));
30    startActivity (callIntent); // privilege escalation vulnerability
31  }
32 }
```

Listing 2: Vulnerable app: receives an Intent and makes a phone call.

The vulnerability occurs on line 30 of Listing 2, where PhoneActivity initiates a system Intent of type `ACTION_CALL`, resulting in a phone call. This is a reserved Android action that requires special access permissions to the system's telephony service. Although PhoneActivity has that permission, it also needs to ensure that the sender of the original Intent message has the required permission to use the telephony 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 15 is commented) to illustrate the vulnerability. If *CallerActivity* does not have the permission to make phone calls (i.e., it is not specified in the corresponding app's manifest file), it is able to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2419611, IEEE Transactions on Software Engineering

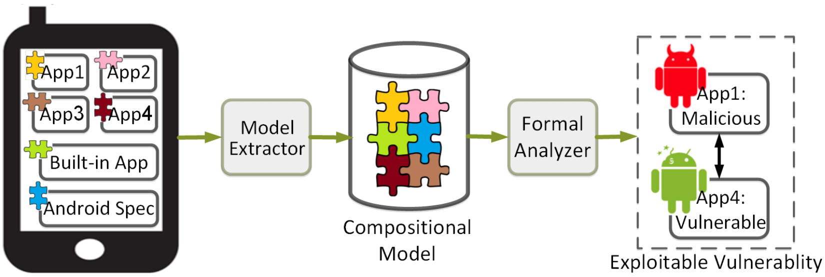

Fig. 1: Overview of COVERT.

make *PhoneActivity* perform that action on its behalf. This is a privilege escalation vulnerability and has been shown to be quite common in the apps on the market [2]. It could be exploited by a malware running on the same phone to call premium-rate numbers.

The above example points to one of the most prominent inter-app vulnerabilities, i.e. privilege escalation, that we take as a running example from a class of vulnerabilities that require compositional analysis to be able to detect effectively.

4 Approach Overview

This section overviews our approach to automatically identify such vulnerabilities that occur due to the interaction of apps comprising a system, and determine whether it is safe for a bundle of apps, requiring certain permissions and potentially interacting with each other, to be installed together. As depicted in Figure 1, COVERT consists of two parts: (1) *Model Extractor* that uses static code analysis techniques to elicit formal specifications (models) of the apps comprising a system as well as the phone configuration; and (2) *Formal Analyzer* that is intended to use lightweight formal analysis techniques to verify certain properties (e.g., known security vulnerability patterns) in the extracted specifications.

COVERT relies on two types of models: 1) *app model* that Model Extractor generates automatically for each Android app; 2) *Android framework spec.* that defines a set of rules to lay the foundation of Android apps, how they behave (e.g., application, component, messages, etc.), and how they interact with each other. The framework specification is constructed once for a given platform (e.g., version of Android) as a reusable model to which all extracted app models must conform. It can be considered as an abstract specification of how a given platform behaves.

Model Extractor takes as input a set of Android application package archives (APK files[1]). To generate the app models, it first examines the application manifest file to determine its architectural information. Besides such high-level, architectural information collected from the manifest file, Model Extractor utilizes static analysis techniques to extract other essential information from the application bytecode. We have built a prototype implementation of the model extractor component on top of *Soot* [11] for static analysis and *Dexpler* [12] for reverse engineering Android APK files. As a result, our prototype implementation of the approach only requires the availability of Android executable files, and not the original source code. COVERT, thus, can be used not only by developers, but also by end-users as well as third-party reviewers to assess the trustworthiness of their mobile devices.

The set of app models extracted in this way are then combined together with a formal specification of the application framework, and checked as a whole for vulnerabilities that occur due to the interaction of apps comprising a system. Finally, a report is returned to the user describing the list of detected vulnerabilities. Upon reviewing the report, end-users and third-party reviewers may choose to protect their devices in a variety of ways, e.g., by disallowing the installation of certain combination of apps, or dynamically restricting certain inter-app communications.

In this research work, we rely on *lightweight formal analysis* techniques [13] for modeling and verification purposes. Such lightweight, yet formally-precise methods, bring fully automated analysis techniques to partial models that represent the key aspects of a system [14]. The analysis is accordingly conducted by exhaustive enumeration over a bounded scope of model instances. These methods thus facilitate application of formal analyzers in development of software-intensive systems. In our prototype tool implementation, we use Alloy [10], as the specification language, and the Alloy Analyzer as the analysis engine. Alloy is a formal specification language based on first order logic, optimized for automated analysis.

Our approach can be applied in an offline setting to determine if a particular configuration for a system comprised of several apps harbors security vulnerabilities. Although not the focus of this paper, we believe the approach could also be applied at runtime to continuously verify the security properties of an evolving system as new apps are installed, and old ones are updated and removed.

[1] APKs are Java bytecode packages used to distribute and install Android applications.

In the following two sections, we describe the details of static analysis used to capture essential application information and formal analysis for verification.

5 Model Extractor

In order to automatically analyze vulnerabilities, we first need a model of each application that would allow us to determine the potential inter-process communications and to also reason about the security properties. In our approach, an app model is composed of the information extracted from two sources: manifest file and bytecode. This section first formally 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* $A =< C, I, F, P, S >$, *where*

- C is a set of components, where each component $c \in C$ has a set of Intent messages $intents(c) \subseteq I$, a set of Intent filters $ifilters(c) \subseteq F$, a set of permissions $perms(c) \subseteq P$ required to access the component c, and a set of sensitive (i.e., security relevant) paths $paths(c) \subseteq S$. Each component is defined as one of the four Android pre-defined component types: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*.
- I is a set of event messages that can be used for both inter- and intra-app communications. Each Intent $i \in I$ has a sender component $sender(i) \in C$, may have a recipient component, 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. If the set $component(i)$ is non-empty, the Intent i is called an *explicit* Intent, as the recipient component is given explicitly.
- F is a set of Intent Filters, where each filter $ifilter \in F$ is attached to a component $c \in C$, and describes an interface (capability) of c in terms of Intents that it can handle. Each $ifilter$ has a non-empty set of $actions(ifilter)$ and two sets of $data(ifilter)$ and $categories(ifilter)$.
- P is a union of required and enforced permissions, $P = P_{Req} \cup P_{Enf}$, where P_{Req} specifies the permissions to which the application needs to have access to run properly and P_{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 $perm_{Used}(c) \subseteq P_{Req}$.
- S is a finite set of vulnerable paths; each path belongs to a component $c \in C$, and is represented as a tuple $< Entry, Destination >$, where *Entry* and *Destination* represent either permission-required APIs or IPC calls.

As shown in Algorithm 1, the Model Extractor performs three major steps to obtain a model of Android app: *Entity Extraction and Resolution* (lines 4–13), *Control Flow Augmentation* (lines 14–16), and *Vulnerable Paths Identification*(line 17). In the first step, the entities are extracted from either the manifest file or the bytecode. Second, COVERT builds an inter-procedural control-flow graph augmented to account for implicit invocations. The generated inter-procedural control-flow graph is further annotated with permissions required to enact Android API calls and Intents. Finally, in the last step, a reachability analysis is performed over the generated graph to determine the exposed components that contain unguarded execution paths reaching permission-required functionalities.

Details of each step, elaborated by Algorithms 2 and 3, are discussed in the rest of this section. To help explain the approach, Figure 2 illustrates the steps of applying our model extraction to the motivating example (cf. Section 3).

Algorithm 1: Model Extractor

```
   Input: app: Android App
   Output: A: App's Extracted Model
1  $A \leftarrow< \{\}, \{\}, \{\}, \{\}, \{\} >$
2  $ICFG \leftarrow \{\}$
3  $summaries \leftarrow \{\}$
   // ▶ Entity Extraction - cf. Sec. 5.1
4  $A.C \leftarrow extractManifestComponents(app)$
5  $A.P \leftarrow extractManifestPermissions(app)$
6  $A.F \leftarrow extractManifestFilters(app)$
7  $IFEntities \leftarrow \{\}$
8  foreach $method \in app$ do
9      $IFEntities \leftarrow identifyIFEntity(method, summaries)$
10 end
11 $resolveIFEntityAttr(IFEntities)$
12 $A.I \leftarrow getIntents(IFEntities)$
13 $A.F \leftarrow getIntentFilters(IFEntities) \cup A.F$
   // ▶ ICFG Augmentation - cf. Sec. 5.2
14 $G \leftarrow constructICFG(app)$
15 $E \leftarrow extractImplicitCallBacks(app)$
16 $ICFG \leftarrow augmentICFG(G, E)$
   // ▶ Vul. Paths Identification - cf. Sec. 5.3
17 $A.S \leftarrow findVulPaths(A.C, ICFG)$
```

5.1 Entity Extraction and Resolution

As part of the entity extraction process, the Model Extractor first identifies the entities comprising the app by parsing and examining the app's manifest files. As shown in Algorithm1 (lines 4–6), it can readily obtain information such as the app's components (C) and their types, permissions that the app requires (P_{Req}), and the enforced permissions ($Perms_{Enf}$) that the other apps must have in order to interact with the app components. It also identifies some of the public interfaces exposed by each application, which are essentially entry points defined in the manifest file through *Intent Filters* (F) of components. However, not all entry points can be extracted from the manifest file, as discussed further below. Figure 2a shows the entities extracted at this stage of analysis corresponding to our running example from Section 3. Although the figure depicts the entities extracted for both apps, the reader should note that in practice COVERT's program analysis runs separately on each app, the results of which are then transformed into separate formal specification modules, as detailed in Section 6.

After collecting these entities through examining the application manifest file, the Model Extractor identifies complementary information latent in the application bytecode. In particular, we also need to extract Intents and Intent Filters, which may be defined programmatically in the bytecode, rather than in the manifest file. Intent Filters for components of type Service and Activity must be declared in their manifest, but for Broadcast Receivers, though, either in the manifest or at runtime.

For each method in an app's component, the algorithm detects and extracts Intents and Intent Filters, as shown in lines 8–10. Android API reference documentation [15] is used in this step to associate specific entities to framework-provided APIs defining or manipulating these entities. In the motivating example (Section 3), samples of entities are identifiable: an `Intent` entity is created in line 12 of Listing 1; the framework API `getIntent` is called in line 13 of Listing 2.

Intents and Intent Filters extracted this way need to be further analyzed to obtain additional information about their attributes. To that end, Model Extractor iterates over each method of the app and calls `identifyIFEntity`, which applies a summary-based iterative data-flow analysis [16] to detect entities and their attributes. For each Intent message, for example, it tracks the message's sender, the target component, the type of action it has (if any), data to be processed by the action, and categories of components that should handle the Intent. Note, however, that the values of attributes are resolved through an additional analysis described later in this section.

`identifyIFEntity` computes a method summary for each analyzed method. The method summary describes information about entities that can be inferred from a method. Method summaries make entity resolution inter-procedural, allowing an entire app to be analyzed. Methods are analyzed in reverse topological order with respect to the app's call graph so that a given method's summary is computed before any methods that call it are analyzed. Cycles in the call graph (e.g., from recursion) are handled in the standard manner, by treating the involved methods as one "super method."

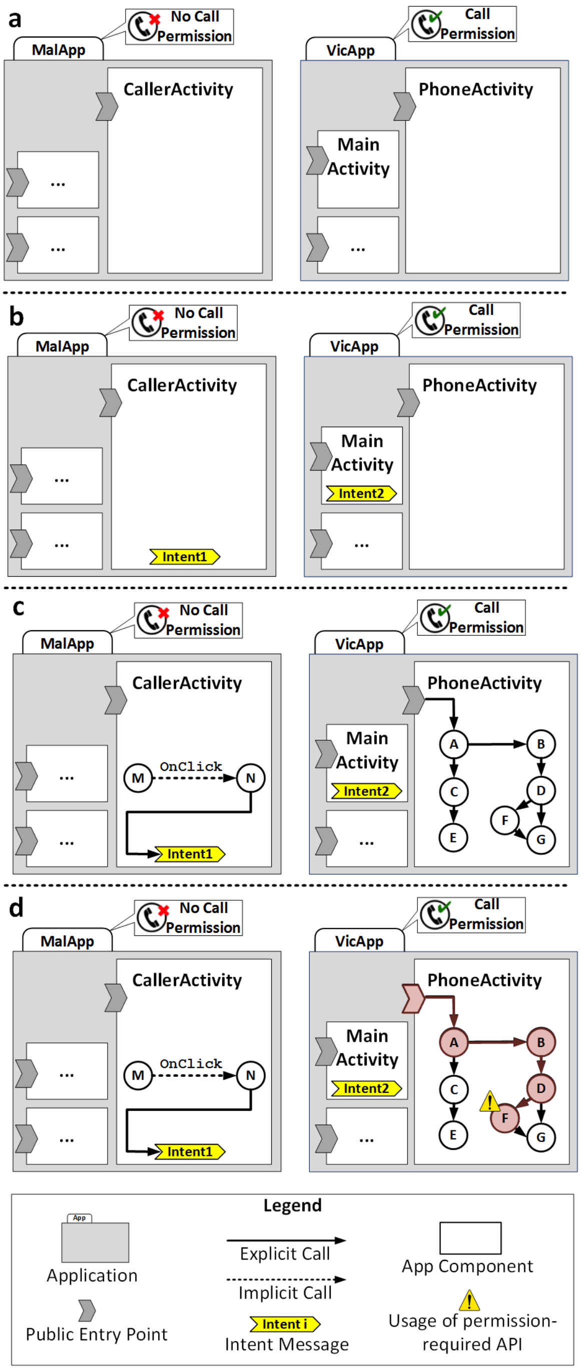

Fig. 2: Extracted models for the apps described in Listings 1 and 2 at different steps of analysis.

The details of `identifyIFEntity` are shown in Algorithm 2. `identifyIFEntity` outputs the set *IFEntities*, which contains identified Intents and Intent Filters that are defined and utilized in the Android app's source code. There are four types of statements that need to be considered to retrieve entity properties: (1) statements that create an entity, (2) statements that set the attributes of an entity, (3) statements that consume an entity, and (4) statements that invoke non-Android API methods.

The first type of statement, handled in lines 10–15 of Algorithm 2, correspond to the APIs creating an entity (e.g., through the constructors). In this case, the newly-created entity is added to the *gen* set in order to be used in the other cases; any entities that are reassigned are added to the *kill* set to prevent further propagation of such entities; and *IFEntities* is updated with the new entity.

The second type of statement, handled by the case of lines 16–19, are the ones that set the attributes of the entity under consideration (i.e., the action, category, data, and target attributes). For example, `Intent.setClassName()` sets the target component for the given Intent.

The third type of statement, handled in lines 20–23 of Algorithm 2, correspond to the APIs that consume entities. Entities are consumed in different ways. An Intent, for example, is consumed when it is sent to a component: `startActivity(Intent)` launches a new Activity by sending an Intent that carries the Activity's description. An Intent Filter, however, is consumed when it is used in registering a Broadcast Receiver. Since the attributes of an entity cannot be set after consumption, the consumed entity is added to the *kill* set.

Finally, for method calls that are not part of the Android API, `identifyIFEntity` utilizes the summary of an invoked method to determine the entities and their attributes that are computed in the method (lines 24–26 of Algorithm 2). In particular, `identifyIFEntity` utilizes the summary of the method invoked in the program statement *stmt* under analysis to update the *gen*, *kill*, and *IFEntities* sets. For example, in line 16 of Listing 2, the non-Android API method `makePhoneCall` is invoked, where a new Intent is created with action and data attributes. `identifyIFEntity` utilizes the method summary for `makePhoneCall` to determine that the invocation of that method results in the creation of a new Intent with action ACTION_CALL and a data attribute. In this case, `updateFromSummary` adds this new Intent to the *gen* and *IFEntities* sets so that the new Intent is recorded and will be propagated by the data-flow analysis. The *kill* set is not modified in this case since the new Intent is not assigned to an already-defined Intent reference.

For aliasing in the case of entities and their attributes, we utilize class hierarchy analysis [16], which produces accurate results for our purposes (as shown in Section 7). However, our algorithm can subsitute the class hierarchy analysis for a more precise analysis (e.g., a points-to analysis), possibly trading off efficiency for precision.

The overall algorithm (line 11 of Algorithm 1) then calls `resolveEntityAttr` to resolve the values associated with the retrieved entity attributes (e.g., the action, categories, and data types of Intents). To do this, it uses string values obtained from string constant propagation [16], which provides a precise solution since, by convention, Android apps use constant strings to define these values. In cases where a string variable's value cannot be determined statically, we take a conservative approach and assume the value to be any string. Despite this conservative approach, our evaluation results (see Section 7) show our technique to be significantly precise, while remaining scalable.

Algorithm 2: identifyIFEntity

```
   Input: $method, summaries$
   Output: $IFEntities$
1  $IFEntities \leftarrow \{\}$
2  $gen[entry] \leftarrow$ {entities passed as parameters to $method$}
3  $workList \leftarrow$ {all statements of $method$}
4  repeat
5      $stmt \leftarrow workList.head$
6      foreach $stmt' \in pred(stmt)$ do
7          $in[stmt] \leftarrow in[stmt] \cup out[stmt']$
8      end
9      switch $stmt.type$ do
10         case Intent or Intent Filter Constructors
11             $entity \leftarrow$ corresponding entity of statement
12             $gen[s] \leftarrow \{entity\}$
13             $kill[s] \leftarrow$ set of reassigned entities
14             $IFEntities \leftarrow \{entity\} \cup IFEntities$
15         end
16         case Entity Attribute Assignment
17             $entity \leftarrow$ corresponding entity of statement;
18             $updateAttr(entity)$
19         end
20         case Intent Sender or Intent Filter Registration
21             $entity \leftarrow$ corresponding entity of statement
22             $kill[s] \leftarrow \{entity\}$
23         end
24         case Non-Android API Method Call
25             updateFromSummary($gen$, $kill$, $IFEntities$,
               $summaries$)
26         end
27     endsw
28     $prevOut \leftarrow out[stmt]$
29     $out[stmt] \leftarrow (in[stmt] \setminus kill[stmt]) \cup gen[stmt]$
30     if $prevOut \neq out[stmt]$ then
31         $workList \leftarrow workList \cup succ(stmt)$
32 until $workList = \emptyset$;
33 summarize($gen$, $kill$, $IFEntities$, $summaries$)
```

It is also possible that a property is disambiguated to more than one value. For instance, consider our running example, the Intent action could be assigned to two different values at runtime, namely "`PHONE_CALL`" and "`PHONE_TEXT_MSG`" defined on lines 6 and 8 of Listing 1, respectively. We take a conservative approach to handle such an issue and generate a separate entity for each of these values, as they contribute different exposure surfaces or event messages in the case of Intent Filters and Intents, respectively.

Figure 2b shows the extracted model corresponding to our running example (recall Section 3) at this stage of analysis. In this particular example, Intents, as well as their properties (not depicted), are the only additional entities extracted from the bytecode. For clarity of presentation, Figure 2b only depicts the Intents relevant to the vulnerability in our example.

5.2 Control Flow Augmentation

Subsequent to extracting entities, Model Extractor needs to determine control flow between methods in order to detect vulnerabilities for privilege escalation. To that end, Model Extractor constructs an inter-procedural control-flow graph (ICFG) of the entire application. An ICFG is a collection of control-flow graphs (CFGs) connected to each other at call sites.

However, due to the event-driven structure of the Android platform, the traditional ICFG generation methods do not connect CFGs at call sites corresponding to implicit invocations. To generate an ICFG that takes implicit invocation into account, we need to include call-backs of an app. These are Android-API methods that no other part of the application explicitly invokes.

To connect the CFGs over implicit calls, we traverse the nodes of each CFG in a depth-first manner, and connect all implicit invocation nodes with the corresponding call-back nodes. For example, in lines 11–15 of Listing 1, an anonymous inner-class is defined within the `onCreate` method to handle the *Click* events triggered by the `btnOk` button. Thus, an edge is added to the app's ICFG from the `setOnClickListener` invocation to the entry point of `onClick`.

Figure 2c shows some parts of ICFGs extracted for each of the apps from Section 3. Here, the dashed line between nodes Ⓜ and Ⓝ indicates an implicit invocation.

5.3 Vulnerable Paths Identification

The last step is to determine if there is a path from each component's IPC entry point to an invocation of a permission-required functionality that is either inappropriately-guarded or unguarded, which may lead to IPC vulnerabilities. For this purpose, COVERT leverages the reachability analysis described in Algorithm 3.

Here, the entry nodes are IPC calls, which represent methods in a component that handle Intents generated by other components or the Android framework itself. Specifically, all app components, including Activities and Services, are required to follow prespecified lifecycles [17] managed by the framework in an event-driven manner. Each component, thus, registers event handlers that serve as the IPC entry points through which the framework starts or activates the component once handled events occur. An Activity, for example, generates a `StartActivity` event that results in another Activity's `onCreate()` method to be called. Moreover, for each entry node, the corresponding component definition in the manifest file is also examined to ensure the component is public (line 5 of Algorithm 3). Recall from Section 2, a component is public, if its specification sets the `EXPORTED` flag or declares Intent filter(s).

The destination nodes are defined as permission-required API calls or Intent messages that are not properly checked. As shown in lines 7–11, to determine destination nodes, for each node in ICFG, `tagCheckedPerm` marks it with two tags: (1) Req_{prm} tag denotes that a statement is called at the node under consideration that requires a particular permission of *"prm"*; and (2) $Check_{prm}$ tag shows the node is guarded by permission *"prm"* checking. Thus, a vulnerable destination node is a node tagged with Req_{prm} but not with the corresponding $Check_{prm}$ tag.

Algorithm 3: findVulPaths

```
   Input: C: set of Components, ICFG
   Output: Vulnerable Paths
1  $Entry \leftarrow \{\}$
2  $Dest \leftarrow \{\}$
3  foreach $c \in C$ do
4      if isPublic(c) then
5          $Entry \leftarrow Entry \cup getEntryPoints(c)$
6  end
7  foreach $n \in ICFG$ do
8      $tagCheckedPerm(n)$
9      if $n.hasTag(Req_{prm}) \wedge !n.hasTag(Check_{prm})$ then
10         $Dest \leftarrow Dest \cup n$
11 end
12 return $pathFinder(Entry, Dest, ICFG)$
```

To identify Req_{prm} tags, `tagCheckedPerm` uses API permission maps available in the literature, and in particular the PScout permission map [18], one of the most recently updated and comprehensive permission maps available for the Android framework. PScout specifies mappings between Android API calls/Intents and the permissions required to perform those calls. The nodes tagged as permission-required are distinguishable in Figure 2d by ⚠ sign. For example, node Ⓕ is a tagged node as it uses *Telephony* API that requires `CALL_PHONE` permission.

Identifying and applying $Check_{prm}$ is trickier, since permission enforcement for a component could be defined at two levels. While the coarse-grained permissions specified in the manifest file are enforced over a whole component, a developer can also add permission checks throughout the code controlling access to particular aspects of a component. The former can be readily checked using the information extracted from the manifest file (recall Section 5.1), but the latter requires further program analysis.

To determine permission-check API invocations that act as guards in code, `tagCheckedPerm` leverages a context-sensitive analysis (i.e., it considers the calling context of a method call) that handles the two most common cases. The first case occurs when a permission-check API is called directly. For the second case, `tagCheckedPerm` determines if a statement invokes a method that results in a call to a permission-check API (e.g., the commented permission check on line 15 of Listing 2). To handle aliasing in this case, `tagCheckedPerm` utilizes class hierarchy analysis, which has proven sufficiently precise for our purposes.

Once entry and destination nodes are identified, *findVulPaths* determines the paths between them (line 12 of Algorithm 3). To achieve high precision in determining paths between entry and destination nodes, our approach is context-sensitive. In the interest of scalability, COVERT's analysis, however, is not path-sensitive (i.e., the analysis does not distinguish information obtained from different paths). The results (see Section 7) indicate no significant imprecision caused by path-insensitivity in the context of Android vulnerability analysis.

Components that contain an $entry \rightarrow destination$ path, returned by *findVulPaths*, are vulnerable to various inter-app attacks. For instance, in Figure 2d the red-colored path of $<$ Ⓐ, Ⓑ, Ⓓ, Ⓕ $>$ is vulnerable, as there is a path from an entry node Ⓐ to an invocation of a permission-required API (i.e., Telephony API). As shown in Listing 1, a malicious app can exploit this vulnerability and call the Telephony API without having the proper privilege.

To achieve scalable, yet precise alias analysis for identifying vulnerable paths, we perform an *on-demand alias analysis* [19]. More specifically, instead of applying the analysis to all variables, only the variables utilized at statements invoking source or sink methods are considered for analysis during vulnerable-path identification.

The Model Extractor produces an extended-manifest file for each Android application. This extended-manifest, documented in an XML format, encompasses all information extracted from both the app bytecode as well as the app manifest file. Once an app model is extracted, it can then be reused for analysis within several bundles of apps. Given a set of extended-manifest files corresponding to a bundle of apps, COVERT generates a package of Alloy modules, which in turn enables their compositional analysis. The next section details the structure of generated Alloy models.

6 Formal Analyzer

In this section we show that our ideas for compositional, formal, and automated analysis of Android apps can be reduced to practice. Our approach automatically transforms the models derived through static analysis into an analyzable specification language, and verifies them against certain properties using the automated analyzers associated with such languages. As an enabling technology, we use the Alloy language [10], to represent a model of Android framework, application models, and to-be-analyzed properties.

There are four main reasons that motivate our choice of Alloy for this work. First, its comprehensible, object-oriented-like syntax, backed with logical and relational operators, makes Alloy an appropriate language for declarative specification of both applications and properties to be checked (i.e., assertions). Second, its ability to automatically analyze specifications with no custom programming is useful as an automation mechanism.

Third, and more importantly, its effective module system allows us to split the overall, complicated system model among several tractable modules. A simple module system is not only convenient, but is an important part of our approach, as it enables effective compositional analysis of, among other things, impenetrable scenarios, where for example a malicious app can leverage a chain of vulnerable components to leak sensitive data or to perform actions that are beyond its individual privileges. Android apps and properties to be checked are strictly separated and modularized in different specifications, which further facilitates reusability of such specifications, and this is clearly where much of the power of our work comes from. Specifically, Android framework specification, application specifications, and specifications of vulnerabilities to be analyzed are all reusable, and this paper shows the promise of paying a one-time cost to formally specify them to enable compositional analysis of Android vulnerabilities.

Lastly, the extraction approach we take in COVERT to generate bundle specifications is incremental. More specifically, the Model Extractor produces a separate extraction-output file for each Android application, independent of other apps in the bundle. The set of extracted app models are then combined together to check for inter-app vulnerabilities. Hence, once an app model is extracted, it can then be reused for analysis within several bundles of apps. That means to add, update or remove an app from the bundle, we only need to add, update, or remove information for that particular app.

To appreciate COVERT's approach, consider that an alternative approach is to detect the inter-app vulnerabilities by performing the program analysis on a whole set of apps simultaneously. But such an approach suffers from two shortcomings. First, it would face serious scalability issues, as a typical mobile device may have tens or hundreds of apps installed on it, and the analysis space grows exponentially with the number of apps to-be-analyzed. Second, it would require such a complex analysis to be performed every time any of the apps are updated, added, and removed. COVERT does not suffer from the same shortcomings because it analyzes the apps in isolation, and relies on the declarative power of formal specification languages (namely Alloy) to separate the various models needed for the analysis, thereby facilitating reuse of the models as well as the results.

In the rest of this section, we first provide a brief overview of Alloy, and then describe how we use it in modeling and thereby analysis of Android applications.

6.1 Alloy Overview

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 [10]. The Alloy Analyzer is a constraint solver that supports automatic analysis of models written in Alloy. The analysis process is based on a translation of Alloy specifications into a Boolean formula in conjunctive normal form (CNF), which is then analyzed using off-the-shelf SAT solvers.

The analyzer provides two types of analysis: *Simulation*, in which the analyzer demonstrates consistency of model specifications by generating a satisfying model instance; and *Model Checking*, which involves finding a counterexample—a model instance that violates a particular assertion. We use the former to compute model instances, represented as satisfying solutions to the combination of models captured from app implementations. This shows the validity of such extracted models, confirming that the captured models are self-consistent, mutually compatible and consistent with the Android specifications modeled in a separate module. The latter is used to verify security properties of interest within the models.

The Alloy Analyzer is a bounded checker, so a certain scope of instances needs to be specified. The scope, for example, states the number of app components. The analysis is thus performed through exhaustive search for satisfying instances within the specified scopes. As a result, the analyzer is sound and complete within such scopes. To take advantage of partial models, the latest version of the analyzer uses *KodKod* [20] as its constraint solver so that it can support incremental analysis of models as they are constructed. The generated instances are then visualized in different formats such as graph, tree representation or XML.

The essential constructs of the Alloy modeling language include: *Signatures*, *Facts*, *Predicates*, *Functions* and *Assertions*. Signatures provide the vocabulary of a model by defining the basic types of elements and the relationships between them. Facts are formulas that take no arguments, and define constraints that any instance of a model must satisfy. Predicates are parameterized and reusable constraints that are always evaluated to be either true or false. Functions 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 is a formula required to be proved. It can be used to check a certain property of a model.

6.2 Formal Model of Android Framework

To carry out the verification analysis, we begin by defining a common Alloy module, *androidDeclaration*, that models 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. It is manually constructed once and does not change, unless there are substantial changes in the way Android operates.

```
1  module androidDeclaration
2
3  abstract sig Application{
4    usesPermissions: set Permission,
5    appPermissions: set Permission
6  }
7  abstract sig Component{
8    app: one Application,
9    intentFilters: set IntentFilter,
10   permissions: set Permission,
11   paths: set Path
12 }
13 abstract sig IntentFilter{
14   actions: some Action,
15   data: set Data,
16   categories: set Category,
17 }
18 fact IntentFilterConstraints{
19   all i:IntentFilter| one i.~intentFilters
20   no i:IntentFilter| i.~intentFilters in Provider
21 }
22 abstract sig Intent{
23   sender: one Component,
24   component: lone Component,
25   action: lone Action,
26   categories: set Category,
27   data: set Data,
28 }
29 abstract sig Path{
30   entry: one Resource,
31   destination: one Resource
32 }
33 abstract sig Permission{}
```

Listing 3: Alloy specifications of essential Android application elements.

Listing 3 partially outlines *androidDeclaration* module, representing Android application fundamentals in Alloy. The essential element types (cf. Def. 1) are defined as top-level Alloy signatures. As mentioned earlier, a signature introduces a basic element type and a set of its relations, called *fields*, accompanied by the type of each field.

There are six top-level signatures to model the basic element types: `Application`, `Component`, `IntentFilter`, `Intent`, `Path`, and `Permission`. Note that these signatures are defined as *abstract*, meaning that they cannot have an instance object without explicitly extending them. Containment relations (e.g., between `Applications` and `Permissions`) are defined as Alloy relations.

According to lines 4–5, the `Application` signature contains two fields of `usesPermissions` and `appPermissions` that identify two sets of permissions, representing P_{Req} and P_{Enf}, respectively (cf. Def. 1).

The `app` field within the `Component` signature (line 8) identifies the parent application that a component belongs to. The keyword ***one*** states that every Component object is mapped to exactly one Application object. Signature declarations of four core component types, namely *Activity*, *Service*, *Receiver* and *Provider*, extend the `Component` signature. In the interest of space, their specifications are omitted from Listing 3.

A component may have any number of filters, each one describing a different interface of the component. Such filters are captured by the `intentFilters` field (line 9). The `permissions` field represents a set of permissions required to access a component. The `paths` field then indicates vulnerable paths within a component.

The `IntentFilter` signature contains three fields of `actions`, `data` and `categories`. The multiplicity keyword ***some*** in Alloy denotes that the declared `actions` relation contains at least one element, and the keyword ***set*** tells Alloy that `data` and `categories` map each `IntentFilter` object to zero or more `Data` and `Category` objects, respectively.

Properties of the IntentFilter signature are declared as a *fact* paragraph (lines 18–21). The ∼ operator denotes the relational inverse operation, forming a new relation by reversing the order of atoms in each tuple of the relation. The statement of line 18, thus, states that each `IntentFilter` belongs to exactly one `Component`. Out of four core component types, three of them can define IntentFilters. To exclude `Content Providers` from having IntentFilters, we add a separate fact constraint specification, represented in line 20.

The `Intent` signature contains five fields of `sender`, `component`, `action`, `data` and `categories`. The first one denotes the component sending the intent. The `component` field identifies the recipient component. The keyword ***lone*** indicates that this element is optional, and an Intent may have one or no declared recipient component. Recall from Section 5, if it maps to a non-empty set, the Intent object is called an *explicit Intent*. The Android intent-resolver delivers explicit Intents to the designated component, without considering other information of the Intent object.

To determine to which component an *implicit Intent*—one that does not specify any recipient component—should be delivered, three elements of `action`, `data`, and `categories` are consulted. The `action` filed names the general action to be performed in the recipient component. The `data` field indicates additional information about the data to be processed by the action, and each Data instance consists of both the URI of the data to be acted on and its MIME media type. Finally, the `categories` field indicates the kind of component that should handle the Intent object. 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.

We define the *entry* and *destination* fields of the `Path` signature based on canonical permission-required resources identified by Holavanalli et al. for Android applications [21]. Examples of entry and destination resources are `NETWORK`, `IMEI`, and `SDCARD`. Among others, the permission `NETWORK`, for example, allows the app to access the Internet, through either WIFI or cellular network. In addition to permission domains, the `IPC` mechanism augments both entry and destination sets, which allows apps to provide services to one another. Figure 3 shows a path identified in *VicApp* with an `IPC` as publicly accessible entry point.

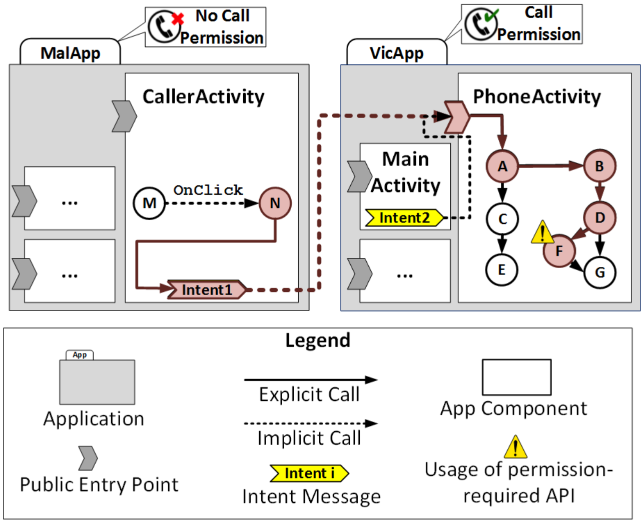

Fig. 3: A vulnerability identified by COVERT for the apps described in listings 1 and 2. The red lines and nodes indicate the vulnerable path.

Finally, the last top-level signature is `Permission`. COVERT captures both the system-defined permissions—declared within the system's Android Manifest—and application-defined permissions—declared within the application manifest file, and documents them as a separate Alloy model shared between Alloy modules of all apps.

6.3 Formal Model of Apps

Three pieces of Alloy specifications are conjoined in the process of modeling various parts of Android apps extracted from their APK files. First, a specification module, called *appDeclaration*, that documents basic element types, such as *Action*, *Category* and *Permission*, shared between Alloy models of all apps. Second, an *app model*, comprising *Components* that constitute the app, *IntentFilters* of each *Component*, as well as required and enforced *Permissions* of the app. This model is represented in a separate Alloy module for each app. Third, an inter-process communication (*IPC*) module that models all Intent messages created within the apps under consideration. All these models rely on the Android framework specification module, presented in the previous Section.

We use snippets of the running example (cf. Section 3) to explain each piece of our formal model. Let us begin with the *appDeclaration* module.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2419611, IEEE Transactions on Software Engineering

```
module appDeclaration

open androidDeclaration

one sig MAIN extends Action{}
one sig CALL_PHONE extends Permission{}
...
```

Listing 4: Part of the declaration of basic element types automatically extracted from Android apps.

Consider the portion of the *appDeclaration* module, shown in Listing 4. At the top, the specification imports the Alloy module for the Android framework. It then declares `MAIN` to be a singleton subset of `Action`. Typically, one activity in an app is specified as the "main" activity, declaring it as the main entry point to the app, and presented to the user when launching the app. In a signature declaration, the keyword ***one*** specifies the declared signature to contain exactly one atom, thereby restricting the signature to be unique. This naming scheme allows us to reuse the term MAIN when we want to declare the main activity of each application. The next statement represents a permission example declared in a similar way. For the sake of clarity, we use the permissions' shorthand in our Alloy model. For example, here we use `CALL_PHONE` to model the particular permission of `android.permission.CALL_PHONE`.

Listing 5 partially delineates the generated specification for the malicious app shown in Listing 1. It starts by importing the *appDeclaration* module (line 3), and then the MalApp is declared as an extension of the `Application` signature. This app does not declare any permission neither as required (`usesPermissions`) nor as enforced (`appPermissions`). The MalApp has a Component of type Activity, named `CallerActivity`, which declares an IntentFilter with `MAIN` and `LAUNCHER` settings, marking it as the main activity of the app.

```
1  module MalApp
2
3  open appDeclaration
4
5  one sig MalApp extends Application{}{
6    no usesPermissions
7    no appPermissions
8  }
9
10 one sig CallerActivity extends Activity{}{
11   app in MalApp
12   intentFilter = IntentFilter1
13   no permissions
14   no paths
15 }
16
17 one sig IntentFilter1 extends IntentFilter{}{
18   actions = MAIN
19   categories = LAUNCHER
20   no data
21 }
```

Listing 5: Part of the generated specification for Malicious app shown in Listing 1.

```
1  module VicApp
2
3  open appDeclaration
4
5  one sig VicApp extends Application{}{
6    usesPermissions = CALL_PHONE
7    no appPermissions
8  }
9
10 one sig PhoneActivity extends Activity{}{
11   app in VicApp
12   intentFilter = IntentFilter2
13   no permissions
14   paths = path1
15 }
16
17 one sig path1 extends Path{}{
18   entry = IPC
19   destination = PHONECALL
20 }
```

Listing 6: Part of the generated specification for Victim app shown in Listing 2.

The code snippet of Listing 6 represents the generated specification for the Victim app shown in Listing 2. The VicApp has access to the `CALL_PHONE` permission (line 6), but declares no permission requirement for other apps to access its own Components (line 7). This app specification then declares the `PhoneActivity` component, exposing a vulnerable path (`path1`) from its entry point to a permission required resource (`PHONECALL`), as represented in Figure 3.

Application interactions in Android occur through Intent messages. We record the interactions among app Components in a separate Alloy module, called *IPC*. The code snippet shown in Listing 7 represents part of the generated specification for the IPC module. After importing modules of the involved apps (lines 3–4), the specification in lines 6–12 models the Intent of Listing 1, where the `CallerActivity` Component sends an explicit Intent (i.e., *intent1* as shown in Figure 3) to the `PhoneActivity` Component, with specified action to be performed and with extra data.

```
1  module IPC
2
3  open VicApp
4  open MalApp
5
6  one sig intent1 extends Intent{}{
7    sender = CallerActivity
8    component = PhoneActivity
9    action = PHONE_CALL
10   no categories
11   extraData = Yes
12 }
13 ...
```

Listing 7: Part of the generated inter-component communication module.

6.4 Checking Android Application Models

The previous sections present a formal model of Android framework (Section 6.2), developed as a reusable Alloy module to which extracted app models conform (Section 6.3). Here, we describe the essence of this work: how one can use the power of proposed formal abstractions to perform the compositional analysis of Android apps.

To that end, we develop assertions that model a set of security properties required to be checked. These assertions express properties that are expected to hold in the extracted specifications. Similar to Android specification, vulnerability assertions are manually constructed once and do not change, unless there are substantial changes in Android that resolve or modify the known types of inter-app vulnerabilities.

Considering the privilege escalation, Davi et al. [22] state it as follows: *"An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee)."* Listing 8 formally expresses the privilege escalation assertion in Alloy. In short, the assertion states that the `dst` component (victim) has access to a permission (`usesPermission`) that is missing in the `src` component (malicious), and that permission is not being enforced in the source code of the victim component, nor by the application embodying the victim component. Recall from Section 5 that there are two ways of checking permissions in Android.

```
assert  privilegeEscalation{
  no disj src, dst: Component, i:Intent|
    (src in i.sender) &&
    (dst in src.^transitiveIPC) &&
    (some p: dst.app.usesPermissions |
      not (p in src.app.usesPermissions) &&
      not ((p in dst.permissions) ||(p in dst.app.
           appPermissions)))
}
```

Listing 8: privilegeEscalation specification in Alloy.

To address a situation, in which more than two components are involved in the privilege escalation, the assertion relies on the specification of the `transitiveIPC` function, shown in Listing 9. The operator "ˆ" represents transitive closure. The expression `src.^transitiveIPC` thus represents the set of all components reachable from `src` following one or more IPCs. The `transitiveIPC` itself relies on the specification of an `intentResolver` function. The *Component*, *Intent* and *IntentFilter* signatures are specified such that they have all the necessary attributes required for Intent resolution. We thus describe intent-resolver as a function augmenting the aforementioned *androidDeclaration* module. This function takes as input an Intent and returns a set of Components that may handle the Intent under consideration. Given the Intent is explicit, it should be delivered to the recipient identified by the `component` field of the Intent (line 10). Otherwise, the resolver checks Components' IntentFilters to find those whose elements are matched against the given Intent. Specifically, an implicit Intent must pass a matching test with respect to each of the *action*, *data*, and *categories* elements on the IntentFilters bound to a component (as stated in lines 13–16). Seeing that a component can define multiple IntentFilters, an Intent that does not match one of a component's IntentFilters may match another (lines 11–12).

```
1  fun transitiveIPC  : Component -> Component {
2    { src, dst  : Component | some i:Intent, d:Path|
3        (src in i.sender) &&
4        (dst in intentResolver[i]) && some dst.paths
5    }
6  }
7
8  fun intentResolver(i:Intent): set Component{
9    {c:Component|    some i.component
10     implies {c = i.component}
11     else { some f: IntentFilter|
12         f.~intentFilter in c
13         && i.action in f.actions
14         && i.categories in f.categories
15         && (i.data.uri = f.data.uri
16         && i.data.type = f.data.type) }
17    }
18 }
```

Listing 9: IntentResolver and transitiveIPC specifications in Alloy.

If an assertion does not hold, the analyzer reports it as a counterexample, along with the information useful in finding the root cause of the violation. Counterexample is a particular model instance that makes the assertion false. Given our running example, the analyzer automatically generates the following counterexample:

```
... // omitted details of model instances
privilegeEscalation_src={MalApp/CallerActivity}
privilegeEscalation_dst={VicApp/PhoneActivity}
privilegeEscalation_i={intent1}
privilegeEscalation_p={appDeclaration/CALL_PHONE}
```

It states that the `VicApp/PhoneActivity` component has access to the `CALL_PHONE` permission, and is resolved by the formal analyzer as the receiver of *intent1* (as shown by a dashed line in Figure 3), which is being sent by the `MalApp/CallerActivity` component lacking access to the `CALL_PHONE` permission. The generated counterexample confirms that the composition of Victim and Malicious apps could result in privilege escalation.

7 Empirical Evaluation

To assess the effectiveness of our approach in revealing Android inter-app vulnerabilities, we have conducted an evaluation that addresses the following research questions:

RQ1. What is the importance of this research? To what extent are Android apps overprivileged and unsafe due to usage of permission-required APIs?

RQ2. How well does COVERT perform? Does it enable compositional analysis of real-world

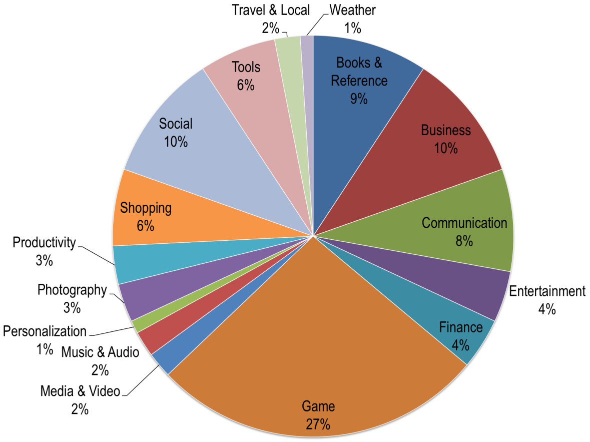

Fig. 4: Distribution of apps selected from the Google Play repository.

Android apps? How much manual effort is involved in the analysis process?

RQ3. What is the overall accuracy of COVERT in detecting inter-app vulnerabilities?

RQ4. How does compositional analysis compare to single app analysis?

RQ5. What is the performance of our prototype tool implemented atop SAT solving technologies and static analyzers?

Our experimental subjects are a set of Android apps drawn from four different app repositories. The first sample set consists of a snapshot of the top 100 popular free apps, available on the Google Play [23] in late November 2013. Our second set of test subjects is representative of open source apps, and includes 300 apps collected from the F-Droid open source repository [24]. To cover the apps available in third-party repositories, we also included 50 apps from Bazaar [25], a local app store, as the third set. The fourth one is a collection of 50 malicious apps identified by the MalGenome project [26].

Figure 4 illustrates the distribution of apps from Google Play repository that were used in our experiments, showing that they are sufficiently diverse, from different categories, and representative of what one can find installed on a typical device. For brevity, we do not show the distribution of apps from other repositories that have different set of categories, but the apps selected from these other repositories were similarly diverse.

To answer RQ1, we examine all of the aforementioned subject apps, to obtain some evidence as to the likelihood of encountering privilege escalation vulnerability in the apps that are available in such markets (§ 7.1).

To address RQ2, we partition the set of apps under study into 10 bundles, each containing 50 apps from three repositories, except the last bundle whose apps are only from the open source repository to enable manual analysis. These bundles simulate collections of apps installed on end-user devices, and we use them to conduct 10 independent experiments. We then report and analyze the experimental results (§ 7.2).

To evaluate the accuracy of warnings reported by COVERT (RQ3), we randomly select 50 apps from the F-Droid open source apps and run our prototype tool on them. We then manually analyze each warning to detect the rate of tool error, i.e., false positive (§ 7.3).

To address RQ4 (single vs. compositional app analysis), we adopt a set of practical security rules, called *Kirin* rules, for Android apps from Enck et al. [6], and formally model each of these rules in such a way that enables their applications for both “compositional” analysis as well as analysis of each “single” app in isolation. We then analyze all the apps in the Malgenome repository against these rules, and compare the results of single and compositional app analysis (§ 7.4).

To address RQ5 (performance benchmarks), we measure the computation time required for both model extraction and formal analysis activities (§ 7.5).

We use the COVERT apparatus we developed based on the approach for carrying out the experiments. COVERT is implemented as a publicly available tool[2]. We have built a prototype implementation of the model extractor component on top of the *Soot* [11] static analysis tools. Soot is developed for analyzing Java bytecode [11]. We thus first use the *Dexpler* transformer [12] to translate Android's dalvik bytecode into the Soot's intermediate representation language, called *Jimple*. As a result, our prototype implementation of the approach only requires the availability of Android executable files, and not the original source code. COVERT, thus, can be used not only by developers, but also by end-users as well as third-party reviewers. The translation of captured app models into the Alloy language is implemented using the *FreeMarker* template engine [27].

7.1 Significance of Compositional Analysis

Table 1 outlines the amount of permissions requested by apps in each repository, along with the fraction of which is actually *used* through API calls, as well as enforced—depicted as *checked* in Table 1—by the apps. Based on the permission map provided by Au et al. [18], we analyzed the fraction of permissions actually needed for API calls performed by the apps under consideration (cf. Section 5). The result shows that overall 32% of acquired permissions are necessary for API calls. This confirms previous studies that showed many Android apps on the market are overprivileged [5], [18]. Applications having extraneous permissions violate the least privilege principle. We also analyzed what fraction of the obtained permissions are checked either within the app manifest file or throughout the code. The difference between the

[2] Research artifacts and experimental data are available at http://www.sdalab.com/projects/covert

TABLE 1: Summary of statistical information about Permissions in subject systems.

Repository	Permissions	
	used	checked
GPlay	1472	
	364 (%24.7.1)	156 (%10.6)
F-Droid	1031	
	505 (%49.0)	77 (%7.5)
MalGenome	499	
	100 (%20.0)	5 (%1.0)
Bazaar	305	
	105 (%34.4)	16 (%5.2)

set of used and checked permissions are important for privilege escalation. The extraneous permissions that result in overprivilege are not susceptible to privilege escalation, unless they are actually used by the permission holders. On average, each app has about 2 unchecked but used permissions that could lead to exploitable vulnerabilities. Indeed, such an unsafe use of permission-required APIs may lead to an exploitable vulnerability provided that there is a path from the exported interface of the app component to the API use. This analysis is the subject of next section.

7.2 Automated Analysis of Applications

The aim of RQ2 is to evaluate the automation level when using COVERT for compositional analysis of real-world Android apps, and how much manual effort is involved in the analysis process. To that end, we evaluate COVERT on bundles of real-world Android apps to determine its ability to detect inter-app vulnerabilities for privilege escalation. Table 2 summarizes the statistical results obtained through running COVERT on Android app bundles. The total number of components defined by the apps in each bundle is shown in the second column. Overall, `Activities`, `Services`, `Broadcast receivers`, and `Content providers` account for 73%, 11%, 12% and 2% of components, respectively.

The *Intents* column delineates the fraction of implicit/explicit Intents out of total Intents in each bundle; on average, about 40% of Intents are implicit, showing that developers, by and large, make inter-component communications explicit. This is promising as there is no guarantee that the implicit Intent will be received by the intended recipient. The next column represents the number of components' interfaces described in terms of Intent filters.

The *Exposed* column shows the number of component surfaces and permissions unsafely exposed to other applications. On average, COVERT detects 5 exposed components in each Bundle. Such components have defined Intent filters that make the components accept incoming Intents, but do not properly enforce access permission, neither in the manifest file nor in the source code. The last column then presents the total number of warnings generated by COVERT for applications of each bundle, and each one represents a unique combination of source and destination components that can lead to a privilege escalation.

Note that reported warnings are about potential security issues. As with other techniques relying on static analysis, our approach is subject to false positives, which could be due to two types of failures in model extraction:

- Strings are used extensively as identifiers in Android apps. Intent properties such as actions, data types, and permissions are all constructed from strings, as shown in our examples. Such strings could also be altered by stateful operations, such as the `append` method, which makes their accurate value elicitation quite challenging. In case an ambiguous value is encountered, during the entity resolution step (Section 5.1), COVERT takes a conservative approach, and considers all possible assignable values.
- COVERT performs reachability analysis (Section 5.3) to determine the permissions actually used by each component, thus ignoring permissions that are obtained, but not used. Yet, there is a possibility that at run-time the permission-required API call or System Intent is not actually reached due to some conditional statements, for example.

The conservative approach we take to deal with non-determinism thus may introduce unnecessary false positives. Encouragingly, this automated analysis still results in a substantial reduction in subsequent manual analysis. Specifically, less than 1% of application components (cf. Table 2, exposed components vs. total components) require further analysis by users. Also, the limitations of the static analysis with respect to, among other things, dynamically loaded code could lead to false negatives as well. To facilitate the process of manual analysis, COVERT provides the location of the potential vulnerability (i.e., filename and method) within the source code.

The results also confirm that an approach combining static analysis and model checking is effective in compositional analysis of Android apps. In this particular case, the reported vulnerabilities provide crucial clues to the security analyst tasked with assessing the security properties of a complex system. Such analysis is not possible with state-of-the-practice tools (e.g., Fortify) that analyze the source code of an application in isolation.

In the next section, we interpret the results through manual analysis of a bundle of open-source applications.

TABLE 2: Summary of experimental results obtained from running COVERT over App bundles.

	Components				Intents		Intent	Exposed		Total
	Activities	Services	Receivers	Providers	explicit	implicit	Filters	Comps	Perms	Warnings
Bundle 1	691				456		169	5	10	34
	511 (%73.95)	70 (%10.13)	91 (%13.17)	19 (%2.75)	300 (%65.79)	156 (%34.21)				
Bundle 2	603				432		148	7	2	16
	434 (%71.97)	76 (%12.6)	78 (%12.94)	15 (%2.49)	302 (%69.91)	130 (%30.09)				
Bundle 3	592				312		185	4	3	25
	425 (%71.79)	65 (%10.98)	85 (%14.36)	17 (%2.87)	218 (%69.87)	94 (%30.13)				
Bundle 4	582				366		191	4	9	32
	423 (%72.68)	75 (%12.89)	71 (%12.2)	13 (%2.23)	232 (%63.39)	134 (%36.61)				
Bundle 5	695				803		359	2	15	16
	569 (81.88)%	61 (8.78)%	53 (7.63)%	12 (1.73) %	408 (50.81) %	394 (49.07) %				
Bundle 6	553				528		225	2	10	15
	445 (80.48)%	52 (9.41)%	47 (8.5)%	9 (1.63) %	278 (52.66) %	249 (47.16) %				
Bundle 7	352				577		137	4	16	35
	242 (68.75)%	43 (12.22)%	62 (17.62)%	5 (1.43) %	349 (60.49) %	227 (39.35) %				
Bundle 8	682				1209		175	3	5	22
	556 (81.53)%	57 (8.36)%	62 (9.1)%	7 (1.03) %	728 (60.22) %	480 (39.71) %				
Bundle 9	511				622		231	5	7	22
	358 (70.06)%	71 (13.9)%	75 (14.68)%	7 (1.37) %	251 (40.36) %	370 (59.49) %				
Bundle 10	660				527		132	5	9	30
	496 (%75.15)	67 (%10.15)	68 (%10.3)	29 (%4.39)	347 (%65.84)	180 (%34.16)				

7.3 Manual Analysis

We selected 50 applications from the F-Droid open source repository, and then manually inspected COVERT's warnings for these applications to evaluate how many warnings correspond to real exploitable vulnerabilities. Statistics of the selected app set are provided as `Bundle 10` in the Table 2. More details about the apps, including their name and model can be found on the project site[3]. In this section, we present the findings of our manual analysis and discuss three representative examples in detail.

COVERT generated 30 warnings for the 50 applications. We manually reviewed all and categorized them according to the classification provided by Chin et al. [2], where each warning is classified as a vulnerability, not a vulnerability, or undetermined. We define a vulnerability to be a component lacking a particular permission getting access to a functionality requiring that permission through an interface exposed by a vulnerable component. In order to detect vulnerabilities, we reviewed the application source code of both sides (sender and destination) for each warning.

Among the 30 reported warnings, we discovered 18 definite vulnerabilities. This represents a 60% true positive rate, which is superior to the prior technique [2], that aimed to identify inter-app vulnerabilities by analyzing the source code of each app in isolation, with a true positive rate of 27.6%. More interestingly, of the 5 application components identified as exposing permissions, all contain at least 1 exploitable vulnerability.

In the rest of this section, we describe a few representative applications and the vulnerabilities we discovered in them.

Case 1: Aard Dictionary → Podax.

The first app is `Aard Dictionary`, a simple dictionary and an offline Wikipedia reader. It defines a *WebViewClient* interface for handling incoming urls, and creates and sends an implicit Intent with the `VIEW` action, should the scheme of the given url matches with one of the specified schemes, such as *http*, *https* and *ftp*.

On the other hand, the app bundle contains the `Podax` app, a podcast downloader and player application. This app accepts Intents with the `VIEW` action, and `http` scheme, which in turn can lead to message passing between the two apps. While the first app that sends the Intent does not have the `INTERNET` permission, the recipient app (`Podax`) has. In addition, the `Podax` app does not check whether the caller has the appropriate permission. This combination, thus, gives rise to a privilege escalation vulnerability.

The sender app here is benign, but if it was malicious it could use the other app's unprotected capability, which may lead to some security risks, such as phishing, by bringing up a web page and enticing the user to enter payment or other private information.

Case 2: Binaural beats therapy → Ermete SMS.

[3] http://www.sdalab.com/projects/covert

`Ermete SMS` is a free web-based text messaging application that has `WRITE_SMS` permission. An Activity component of this application exposes an unprotected interface that receives Intents with `SEND` action. Upon receiving an Intent, the `ComposeActivity` component extracts the payload of the given Intent, and sends that data via text message to a number specified in the payload, without checking the permission of Intent sender.

The other app, `Binaural beats therapy`, is designed for relaxation, creativity and many other desirable mental states. This app does not have the `WRITE_SMS` permission, but it sends an Intent with `SEND` action and *text/plain* payload data, which could be received by the first app. This case represents a false positive as the Intent sent by the `Binaural beats therapy` app does not actually contain the fields required by `Ermete SMS` to send a text message, but points to an important security risk, where a malicious app could use the exposed messaging service.

Case 3: PurpleDock → RMaps.

`RMaps` is an on- and off-line navigation tool. In addition to GPS permissions like `ACCESS_FINE_LOCATION`, it has `INTERNET` permissions to work with online maps such as Google and Microsoft maps. This application exposes an activity, which receives `VIEW` Intents with `geo` scheme, a URI scheme for geographic locations. On the other hand, `PurpleDock` is a simple app that automatically turns on when the handset is placed into the car mount, and provides navigation as one of its features.

`RMaps`'s geo Intents are intended for internal use, and other applications, including PurpleDock that sends a geo message via Intent, should not be able to control locations shown by the app interface. However, with the current implementation, as it does not check the permission of Intent senders, the exposed component can be manipulated by a malicious application for GPS spoofing (i.e., display a wrong location).

7.4 Compositional vs. Single App Analysis

Enck et al. [6] provide a set of practical security rules, called Kirin rules, to prevent malwares from exploiting Android applications. Each rule represents undesirable security properties in terms of the configuration available in manifest files. Kirin rules, thus, decide whether the security configuration bundled with a single app is safe or not, but they do not consider the case in which malicious apps collude to combine their permissions, allowing them to perform actions beyond their individual privileges.

To analyze these rules using our approach, we formalized them in Alloy. Each rule is modeled as an assertion to be analyzed independently. We also developed a compositional version of each rule, leveraging the privilege escalation predicate. This in turn enabled us to apply the two sets of rules and compare the results of isolated analysis versus compositional analysis.

To make the idea concrete, we illustrate one of these rules along with its formal representations for both compositional and single app analysis. Consider the following Kirin security rule (KSR 6): *"An application must not have `RECEIVE_SMS` and `WRITE_SMS` permission labels [6]."*

Listing 10 partially outlines the two Alloy assertions specified to check the rule against either (a) a single app or (b) a combination of apps that may collude to combine their permissions. Assertion (a) states a direct representation of the aforementioned rule in Alloy, while assertion (b) restates the same rule against multiple apps. It uses the `isPrivilegeEscalation` predicate (line 16) to check the occurrence of privilege escalation between the two apps with respect to the `p2` permission. The `p1` and `p2` permissions could be either `RECEIVE_SMS` or `WRITE_SMS` (lines 11–12), but they should be distinct as enforced by *disj* keyword (line 11). The predicate takes as input two components `c1` and `c2`, an Intent, and a permission. The `c1` component belongs to the `app1` and `c2` to the `app2`, omitted in Listing 10 (b) in the interest of space. The assertion then at the very end of line 16 checks the case in which one app contains both permission labels. Note that in practice developing two different assertions is not necessary as the latter, in effect, covers the former. Here, we developed the former for experimental purposes, and to compare the results of single versus compositional analysis.

We analyzed all the apps in the Malgenome repository against each of these rules. Table 3 summarizes the results. Rows represent Kirin security rules that

```
1  // (a) single app analysis
2  assert KirinRule6{
3    no p1,p2: Permission| {some app: Application|
4    (p1 = RECEIVE_SMS) and (p2 = WRITE_SMS) and
5    (p1 in app.usesPermissions) and (p2 in app.
         usesPermissions)
6    }
7  }
8  ------------------------------------------
9  // (b) compositional app analysis
10 assert KirinRule6_Compos{
11  no disj p1,p2: Permission| {some app1,app2:
         Application,
12  c1,c2:Component, intent1:Intent|
13  (p1 in RECEIVE_SMS+WRITE_SMS) and
14  (p2 in RECEIVE_SMS+WRITE_SMS) and
15  (p1 in app1.usesPermissions) and (p2 in app2.
         usesPermissions)
16  and (isPrivilegeEscalation[c1, c2, intent1, p2]or(
         app1 = app2))
17  }
18 }
```

Listing 10: Specification of a Kirin rule for (a) single and (b) compositional app analysis.

TABLE 3: Compositional vs. Single App Analysis of Kirin Rules over the Malgenome app repository.

Sec. Rule	Sing. App. Analysis	Compositional Analysis
KSR 1	-	-
KSR 2	-	-
KSR 3	2	2
KSR 4	2	8
KSR 5	2	11
KSR 6	10	14
KSR 7	11	14
KSR 8	3	3
Overall	30	52

TABLE 4: Experiments performance statistics.

	Construction Time (Sec)	Analysis Time (Sec)
Bundle 1	412	252
Bundle 2	226	123
Bundle 3	441	65
Bundle 4	158	57
Bundle 5	191	239
Bundle 6	88	85
Bundle 7	120	123
Bundle 8	350	374
Bundle 9	295	299
Bundle 10	204	45

we formally modeled in Alloy to be analyzed using our approach. Columns represent the analysis type, either single app analysis (as performed by the Kirin tool [6]) or compositional analysis. Each cell indicates the number of vulnerabilities detected. As we can see, the compositional rule analysis detects more vulnerabilities, without missing any vulnerability identified by single app analysis. The experimental results indicate the overall improvement of 73% in detecting vulnerabilities using a compositional analysis approach.

7.5 Performance and Timing

The final evaluation criteria are the performance benchmarks of model extraction and formal analysis activities. We used a PC with an Intel Core i7 2.4 GHz CPU processor and 8 GB of main memory, and leveraged Sat4J as the SAT solver during the experiments.

Compositional Analysis of Android apps using our approach consists of three steps: (1) The app models are collected and documented as Alloy specifications. (2) The extracted Alloy models are transformed into 3-SAT clauses using the Alloy Analyzer. (3) An off-the-shelf SAT solver explores the space to find counterexamples. We measured the computation time required for each step separately.

The scatter diagram shown in Figure 5 plots the time taken to analyze the collected apps for model extraction in seconds. The results show that the analysis time scales almost linearly with the size of apps in all three repositories. However, as the set of most popular apps collected from the Google Play repository—represented by dark blue in the diagram—are typically larger than apps from the other two repositories, their model extraction takes more time. According to the diagram, our approach is able to statically analyze and infer specifications for the largest apps in less than three minutes. As our implementation separates model extraction analysis from Alloy model generation, and each app bytecode is analyzed independently (cf. Section 5), the total static analysis time scales linearly with the total size of apps.

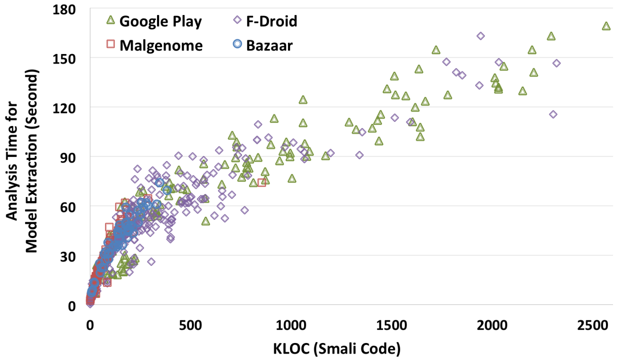

Fig. 5: Scatter plot representing analysis time for model extraction of Android apps.

Table 4 shows the time involved in compositional verification of Android apps (steps 2 and 3). The first column represents the time spent on transforming Alloy models into 3-SAT clauses, and the second the time spent in SAT solving to find all counterexamples for each app bundle. The timing results show that COVERT 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.

8 Discussion and Limitations

There is a growing need for technologies that can support the security analysis of complex systems in a compositional manner, whereby the security of a system is reasoned about in terms of the security properties inferred from its constituents. We argue this is the holy grail of software security analysis research. For the security analysis techniques to scale to ever-increasing complex systems, they need to become compositional in nature. COVERT takes an important step towards this overarching objective in the context of Android apps, but we envision the ideas set forth in this research to find a broader application in other computing domains as well.

Note that single app analysis and compositional analysis have their own technical merits. From an application developer's perspective, analyzing each app in isolation may provide sufficient feedback to fix the issues in the code (i.e., remove the vulnerabilities).

```
1  assert appCollusion{
2      no disj cmp1, cmp2: Component|
3          some cmp1.paths && some cmp2.paths &&
4          cmp1.app != cmp2.app &&
5          match[cmp1.paths.destination,cmp2.paths.entry]
6  }
7
8  pred match(pathSink: set Resource+Intent,
9    pathSource: set Resource+IntentFilter){
10     SDCARD in pathSource & pathSink ||
11     LOG in pathSource & pathSink ||
12     (some i:Intent, f: IntentFilter|
13     i in pathSink && f in pathSource && matchIPC[i,f])
14 }
```

Listing 11: Specification of the application collusion vulnerability in Alloy.

On the other hand, when the purpose of analysis is to assess the trustworthiness of a system, comprised of multiple proprietary apps that may interact with one another, compositional analysis is needed to detect vulnerabilities that may exist in the system. One can imagine an organization may need to use a tool such as COVERT to analyze the security properties of apps deployed on phones assigned to its employees. Such an organization may not be in a position to fix the issues in the apps, as the apps may be proprietary, but it can control the apps that are installed on the devices.

Our analysis indicates that IPC vulnerabilities are ubiquitous, and demonstrates why prior techniques relying only on single app analysis are insufficient for detecting such vulnerabilities. Our experiences with a novel approach for compositional app analysis and its evaluation in the context of hundreds of real-world Android apps collected from variety of repositories have been very positive. The experimental data shows that COVERT can effectively detect such inter-app vulnerabilities in the order of few minutes.

8.1 Development Effort

The framework specification is not expected to be written by individual users of COVERT, rather by the provider of the framework or COVERT. The specification for a framework, such as Android, is developed once and can be reused by others. Thus, it poses a one-time cost, and the required effort depends on the level of familiarity with the framework and the specification language. Using executable specification languages, one can also immediately check the correctness of even partial specifications. In our own experience, Alloy helped us to find errors early in specifying formal semantics. More specifically, during the modeling process, its analyzer performed syntactic checks to expose, for instance, inaccurate use of signatures (such as accessing a nonexistent field of a signature). We also used the analyzer to check the conformance of automatically generated models of apps derived through static analyzer to the framework meta-model.

8.2 Other Types of Vulnerabilities

While privilege escalation vulnerability has been the focus of our research, we believe COVERT can be extended, and significant components of it reused, for detecting other types of inter-app vulnerabilities. For instance, an important class of inter-app vulnerabilities are due to information leakage. For these types of vulnerabilities, COVERT's program analysis needs to be extended to take information flow into account for Android apps. While not the focus of this paper, in an alternative configuration, we augmented COVERT's reachability analysis described in Section 5.3 with a taint flow analysis approach (see [28]) to detect possible information leaks between apps.

We illustrate the reuse and extension potential of COVERT through an example of the application collusion vulnerability. Consider two applications `A` and `B`; `B` reads data from a particular folder in SD card and sends the data out through Internet, and `A` writes data to the folder that `B` reads from. Since `B` does not expose the sending action through its interface (Intent-Filter), it cannot be detected by the *privilegeEscalation* check, specified in Listing 8.

To extend COVERT for supporting the analysis of this scenario, the only thing required is to model it as an assertion, expressing properties to be checked in the extracted specifications. Listing 11 expresses such an assertion for the application collusion. The assertion states that there are two components in different applications; each contains a sensitive data flow path, where the sink of one matches the source of the other. Recall from Section 6 that the `paths` field denotes information paths between permission domains for each component.

Continuing with our example, the apps `A` and `B` contain the flow permissions: `IMEI` → `SDCARD` and `SDCARD` → `NETWORK`, respectively. These two paths will set the *match* predicate to be true (line 8), and thus COVERT identifies it as an instance of the application collusion. Note that since applications specifications and properties to be checked are strictly separated, arbitrary vulnerabilities can be detected with minimal effort.

8.3 Limitations

There are of course limitations in our approach. Similar to any approach based on static analysis, our approach is subject to false positives. We believe a fruitful avenue of future research is to complement COVERT with dynamic analysis techniques. In principle, it should be possible to leverage dynamic analysis techniques to automatically confirm some of the vulnerabilities (e.g., by executing the vulnerable code), further reducing and targeting the manual analysis effort.

Another advantage of dynamic analysis is that it can be used to address vulnerabilities in native code.

Android Apps may include native code in addition to Java code, in the form of a Java Native Interface (JNI) library. Although native code is also obligated to the permission system [5], it may dynamically load code, which cannot be sufficiently addressed through static analysis techniques.

Through dynamic analysis it would also be possible to address upcoming features, such as Google *App Ops*, which provides a Permission Manager that allows users to revoke granted permissions after installation time. This feature was introduced in Android 4.3, but subsequently removed in Android 4.4.2 [29]. Given that granular revocation or allowance of permissions after installation time is a dynamic mechanism, a dynamic analysis would be needed to handle possible vulnerabilities arising from this feature. For example, COVERT's incremental, compositional analysis for permission leakages can be performed on a permission-modified app and related apps at runtime, with appropriate instrumentation or monitoring capabilities added.

In addition to Intent-based event messaging as one type of IPC mechanism, Android applications can communicate through remote procedure calls (RPC) by method-invocation interaction using stubs, which are automatically generated from the specification of a component's interfaces in Android's Interface Definition Language (AIDL). While the focus of this paper is on the inter-app vulnerabilities that arise due to Intent-based event messaging—shown to be the primary IPC mechanism in Android—supporting other IPC mechanisms constitutes an interesting avenue for our future work.

This paper provides substantial supporting evidence for analyzing one of the most significant inter-app vulnerabilities, i.e. privilege escalation. It would be interesting to see how our approach fares when applied to other types of inter-app vulnerabilities [2], [8], [30], which forms a thrust of our future work.

9 Related Work

Android security has received a lot of attention in recently published literature, due mainly to the popularity of Android as a platform of choice for mobile devices, as well as increasing reports of its vulnerabilities [1], [30]. Here, we provide a discussion of the related efforts in light of our research.

9.1 Android Program Analysis for Security

A large body of work [2], [3], [31]–[35] focuses on performing program analysis over Android applications for security, which can be categorized based on their underlying static or dynamic analysis technique.

Chin et al. [2] studied security challenges of Android communication, and developed ComDroid to detect those vulnerabilities through static analysis of each app. Octeau et al. [33] developed Epicc for analysis of Intent properties—except data scheme—through inter-procedural data flow analysis. FlowDroid [28] introduces a precise approach for static taint flow analysis in the context of each application component. CHEX [36] also takes a static method to detect component hijacking vulnerabilities within an app. We share with this approach the emphasis on separating model extraction from vulnerability analysis, enabling extension/revision of each, independent of the other. However, these research efforts, like many others we studied, are mainly focused on Intent and component analysis of one application. COVERT's analysis, however, goes far beyond single application analysis, and enables compositional analysis of the overall security posture of a system, greatly increasing the scope of vulnerability analysis. Doing this requires application of verification techniques in a way scalable to handle analysis of complex systems comprising multiple apps interacting with each other. COVERT, to our knowledge, is the first tool with this capability.

DidFail [37] introduces an approach for tracking data flows between Android components to detect potential data leaks. However, it does not target the problem we are addressing, namely detecting the permission leakage. Moreover, similar to many other techniques we studied, DidFail is a purely program analysis tool, and does not incorporate a formal verification technique.

Along the same line, AndroidLeak [31] statically analyzes information leak in Android. Its analysis does not cover Intents, nor cross-application flows. SCanDroid [38] statically analyzes data flows to detect permission inconsistencies between applications that could possibly allow malicious access to sensitive information. It requires the source code of applications, and has never been evaluated over real-world applications. Mann and Starostin [34] also developed a framework to detect privacy leaks from the Android APIs. Similar to ScanDroid, this framework was never tested against real-world applications. Zhou and Jiang [35] analyzed vulnerabilities that are due to the existence of unprotected content provider components. While this work is concerned with the potential risks of passively leaking content, it does not consider the problem that we address, the automation of inter-app vulnerability analysis.

Apart from techniques based on static analysis, several tools use dynamic analysis to detect vulnerabilities in smartphone applications. TaintDroid [3] detects information leak vulnerabilities using dynamic taint flow analysis at the system level. IPC Inspection [39] prevents privilege escalation at OS level. Recipients of IPC requests are re-instantiated according to the privileges of their callers, guaranteeing that the callee does not have privileges more than that of the caller. However, maintaining multiple instances of applications with modified privileges imposes a notable performance overhead. Saint [40] analyzes configuration and runtime behavior of Android apps to enforce security policy and to allow only legitimate permissions.

These research efforts share our emphasis on leveraging program analysis to capture some information from application implementations. However, our work differs in several ways. First, our approach is geared towards the application of formal techniques to verify certain properties in Android applications. A novel contribution of our work is the ability to bridge from application implementations to formal specifications using static code analysis techniques. Second, previous studies of Android applications analyze a single app in isolation. Our modular approach can be used to greatly increase the scope of application analysis by inferring the security properties from individual apps and checking them as a whole for vulnerabilities that are due to the interaction of apps comprising a system. Third, many of the previously proposed solutions [3], [6], [9], [39] require changes to one or more components of the Android middleware, such as Application Installer, Reference Monitor, and Dalvik Virtual Machine. Our approach, in contrast, requires no platform modifications.

9.2 Android Permissions

The other relevant line of research focuses on Android's permissions and their use across applications [6], [21], [41]–[45]. Barrera et al. [41] examined permission requirements over a set of 1,100 Android applications to analyze how permissions are used in such applications. Their result shows that a small fraction of permissions are extensively used. Kirin [6] 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. Whyper [45] is a tool that checks the app's requested permissions against its description, thereby enabling the user to determine if certain requested permissions are suspicious. Vidas et al. [44] have developed a tool that scans the Android documentation to extract permission specifications. These techniques typically rely only either on the Android documentation or permission requests specified within the application manifest, rather than analyzing the code to check whether or how such permissions are used by applications.

Along the same line, another thrust of research statically analyzes the apps source code to study their permission use. Among others, Felt et al. [5] have developed *Stowaway*, a tool for performing an over-privilege analysis on application source code. Applying automated testing techniques on the Android API, they developed a set of permission maps—documenting which APIs require what permissions—used in detecting overprivilege. Similarly, Au et al. [18] have developed *PScout* to extract the permission specification from the Android OS source code using static analysis, which led to a comprehensive set of permission maps for Android. We used *PSCout*'s permission map in our tool implementation to analyze whether applications under consideration properly check permissions before calling APIs, thereby reducing false positives in COVERT.

9.3 Formal Approaches

The other relevant thrust of research has focused on formal modeling and automated verification of software applications. Fragkaki et al. [7] proposed a formal framework as an extension to the Android permission mechanism. Chaudhuri [46] also proposed a formal language to describe applications and a type system to reason about information flows. This work, however, does not provide any implementation for the proposed approach. Martin et al. [47] developed PQL, which provides a specification language for querying Java applications to detect errors and security flaws. PQL does not include mechanisms for handling Intents, which require a flow-sensitive analysis; the Android lifecycle; and bundles of applications. Thus, PQL focuses on single applications, while COVERT focuses on compositional analysis. Alloy also has been widely used for modeling and analysis in a variety of contexts, including checking code against partial specifications [48]–[50], analysis of software architecture [51], [52], specification based testing [53], and security [54]–[56]. In our prior work, we developed DroidGuard [55], which uses the Alloy analysis engine to generate specifications of Android security policies. These policies are then proactively applied as preventive measures to guard against yet unknown malicious behavior. While DroidGuard targets synthesis and runtime enforcement of security policies, this work addresses precise detection of inter-app vulnerabilities. Chen et al. [56] provided a logical formulation of general security concepts, and modeled it in Alloy. Their model is very abstract, and has not been applied in any particular domain or application. Targeting a real-world banking system, Ramananandro [54] used Alloy to model and check specifications of an electronic smart card system. However, unlike our work, the translation to Alloy is not automated in this research effort. To the best of our knowledge, COVERT is the first formally-precise analysis technique leveraging Alloy for automated compositional verification of Android apps.

10 Conclusion

This paper presents a novel approach for compositional analysis of Android inter-app vulnerabilities. Our approach employs static analysis to automatically recover models that reflect Android apps and interactions among them. It is able to leverage these models to identify vulnerabilities due to interaction of multiple apps that cannot be detected with prior techniques relying on a single app analysis. We formalized the basic elements of our analysis in an analyzable specification language based on relational logic, and developed a prototype implementation, COVERT, on top of our formal analysis framework. The experimental results of evaluating COVERT against privilege escalation—one of the most prominent inter-app vulnerabilities—in the context of hundreds of real-world Android apps corroborates its ability to find vulnerabilities in bundles of some of the most popular apps on the market.

11 Acknowledgment

This work was supported in part by awards D11AP00282 from the US Defense Advanced Research Projects Agency, H98230-14-C-0140 from the US National Security Agency, HSHQDC-14-C-B0040 from the US Department of Homeland Security, and CCF-1252644 from the US National Science Foundation.

References

[1] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, C. Glezer, Google android: A comprehensive security assessment, Security & Privacy, IEEE 8 (2) (2010) 35–44.

[2] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in android, in: Proceedings of the 9th international conference on Mobile systems, applications, and services, MobiSys '11, ACM, New York, NY, USA, 2011, pp. 239–252. doi:10.1145/1999995.2000018.

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

[4] P. Hornyack, S. Han, J. Jung, S. Schechter, D. Wetherall, 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.

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

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

[7] E. Fragkaki, L. Bauer, L. Jia, D. Swasey, Modeling and enhancing android's permission system, in: Proc. of ESORICS, 2012.
URL http://link.springer.com/chapter/10.1007/978-3-642-33167-1_1

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

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

[10] D. Jackson, Alloy: a lightweight object modelling notation, TOSEM 11 (2) (2002) 256–290.
URL http://portal.acm.org/citation.cfm?doid=505145.505149

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

[12] A. Bartel, J. Klein, Y. LeTraon, M. Monperrus, Dexpler:converting android dalvik bytecode to jimple for static analysis with soot, in: Proc. of SOAP, 2012.

[13] J. Woodcock, P. G. Larsen, J. Bicarregui, J. Fitzgerald, Formal methods: Practice and experience, ACM Comput. Surv. 41 (4) (2009) 19:1–19:36. doi:10.1145/1592434.1592436.
URL http://doi.acm.org/10.1145/1592434.1592436

[14] P. Zave, A practical comparison of alloy and spin, Tech. rep. (2012).

[15] Android api reference document, http://developer.android.com/reference.

[16] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2Nd Edition), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[17] Android developers guide.
URL http://developer.android.com/guide/topics/fundamentals.html

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

[19] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, S. Guarnieri, Andromeda: Accurate and scalable security analysis of web applications, in: Fundamental Approaches to Software Engineering, Springer, 2013, pp. 210–225.

[20] E. Torlak, A constraint solver for software engineering: Finding models and cores of large relational specifications, PhD thesis, MIT (Feb. 2009).
URL http://alloy.mit.edu/kodkod/

[21] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, L. Ziarek, Flow permissions for android, in: Proceeding of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013.

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

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

[24] F-droid.
URL https://f-droid.org/.

[25] Bazaar.
URL https://cafebazaar.ir//.

[26] Malgenome project, http://www.malgenomeproject.org.

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

[28] S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, 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.

[29] Protecting your privacy: App ops, privacy guard, and xprivacy.
URL http://www.xda-developers.com/protecting-your-privacy-app-ops-privacy-guard-and-xprivacy/

[30] W. Enck, D. Octeau, P. McDaniel, S. Chaudhuri, A study of android application security, in: Proc. of USENIX, 2011.

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

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

[33] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, Y. L. Traon, 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, 2013.
URL http://siis.cse.psu.edu/epicc/papers/octeau-sec13.pdf

[34] C. Mann, A. Starostin, A framework for static detection of privacy leaks in android applications, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC'12, ACM, New York, NY, USA, 2012, pp. 1457–1462.

[35] Y. Zhou, X. Jiang, Detecting passive content leaks and pollution in android applications, in: Proceedings of the 20th Network and Distributed System Security Symposium (NDSS 2013), 2013.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2015.2419611, IEEE Transactions on Software Engineering

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

[37] W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer, 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, SOAP '14, ACM, New York, NY, USA, 2014, pp. 1–6. doi:10.1145/2614628.2614633.
URL http://doi.acm.org/10.1145/2614628.2614633

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

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

[40] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel, Semantically rich application-centric security in android, in: Proc. of the 25th Annual Computer Security Applications Conference (ACSAC), 2009.

[41] D. Barrera, H. Kayacik, P. Oorschot, A. Somayaji, A methodology for empirical analysis of permission-based security models and its application to android, in: Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2010.

[42] Y. Zhou, Z. Y. Wang, W. Zhou, X. Jiang, 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.

[43] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, Riskranker: scalable and accurate zero-day android malware detection, in: Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys 2012), 2012.

[44] T. Vidas, N. Christin, L. Cranor, Curbing android permission creep, in: Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011), 2011.

[45] R. Pandita, X. Xiao, W. Yang, W. Enck, T. Xie, Whyper: Towards automating risk assessment of mobile applications, in: Proceedings of the 22Nd USENIX Conference on Security, SEC'13, USENIX Association, Berkeley, CA, USA, 2013, pp. 527–542.
URL http://dl.acm.org/citation.cfm?id=2534766.2534812

[46] A. Chaudhuri, Language-based security on android, in: Proceedings of Programming Languages and Analysis for Security (PLAS'09), 2009, pp. 1–7.

[47] M. Martin, B. Livshits, M. S. Lam, Finding application errors and security flaws using pql: a program query language, in: ACM SIGPLAN Notices, Vol. 40, ACM, 2005, pp. 365–383.

[48] S. Khurshid, Darko Marinov, D. Jackson, An analyzable annotation language, in: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '02, ACM, New York, NY, USA, 2002, pp. 231–245. doi:10.1145/582419.582441.
URL http://doi.acm.org/10.1145/582419.582441

[49] D. Jackson, M. Vaziri, Finding bugs with a constraint solver, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2000.

[50] J. P. Near, A. Milicevic, E. Kang, D. Jackson, A lightweight code analysis and its role in evaluation of a dependability case, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE'11, ACM, New York, NY, USA, 2011, pp. 31–40. doi:10.1145/1985793.1985799.
URL http://doi.acm.org/10.1145/1985793.1985799

[51] J. S. Kim, D. Garlan, Analyzing architectural styles, Journal of Systems and Software 83 (7) (2010) 1216–1235.

[52] H. Bagheri, K. Sullivan, Monarch: Model-based development of software architectures, in: Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), 2010, pp. 376–390.

[53] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, S. Khurshid, Query-aware test generation using a relational constraint solver, in: Proceedings of the 23rd IEEE/ACM International Conference on AutomatedSoftwareEngineering, pp. 238–247.

[54] T. Ramananandro, Mondex, an electronic purse: Specification and refinement checks with the alloy model-finding method, Formal Asp. Comput. 20 (1) (2008) 21–39.

[55] H. Bagheri, A. Sadeghi, R. Jabbarvand, S. Malek, Automated dynamic enforcement of synthesized security policies in android, Tech. Rep. GMU-CS-TR-2015-5, Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030-4444 USA (2015).

[56] C. Chen, P. Grisham, S. Khurshid, D. Perry, Design and validation of a general security model with the alloy analyzer, in: Proceedings of the ACM SIGSOFT First Alloy Workshop, pp. 38–47.

Hamid Bagheri is a Postdoctoral researcher at the department of Computer Science at George Mason University and the Computer Science and Artificial Intelligence Laboratory at MIT. He received his Ph.D. in Computer Science from University of Virginia, a M.Sc. in Software Engineering from Sharif University of Technology, and his B.Sc. in Computer Engineering from University of Tehran. Hamid is broadly interested in software engineering, and particularly in practical software synthesis and analysis using concepts from fields like formal methods, program analysis, model-driven development, and software architecture. He is a member of the ACM and IEEE.

Alireza Sadeghi is a Ph.D. student in the Computer Science Department at George Mason University. His research interests focus on software engineering, specifically, application of program analysis in security and energy consumption assessment of mobile applications. Sadeghi received the B.S. degree in computer (software) engineering and M.S. degree in information technology from Sharif University of Technology in 2008 and 2010, respectively. He is a member of ACM, and ACM SIGSOFT.

Joshua Garcia is a Postdoctoral Research Fellow in the Computer Science Department at George Mason Universiy (GMU). He received his Ph.D. and M.S. degrees in Computer Science from the University of Southern California (USC), and his B.S. degree in Computer Engineering and Computer Science from USC. Garcia's research interests are in software engineering. His work focuses on software architecture, software maintenance and re-engineering, and software security.

Sam Malek is an Associate Professor in the Computer Science Department at George Mason University (GMU). He is also the director of GMU's Software Design and Analysis Laboratory. His general research interests are in the field of software engineering, and to date his focus has spanned the areas of software architecture, autonomic computing, software security, and software analysis and testing. Malek received his Ph.D. and M.S. degrees in Computer Science from the University of Southern California and his B.S. degree in Information and Computer Science from the University of California Irvine. He has received numerous awards for his research contributions, including the National Science Foundation CAREER award, the GMU Emerging Researcher/Scholar/Creator award, and the GMU Computer Science Department Outstanding Faculty Research Award. He is a member of the ACM, ACM SIGSOFT, and IEEE.