2018 IEEE International Conference on Software Maintenance and Evolution

Efficient, Evolutionary Security Analysis of Interacting Android Apps

Hamid Bagheri*, Jianghao Wang*, Jarod Aerts*, Sam Malek†
*Department of Computer Science and Engineering, University of Nebraska-Lincoln
bagheri@unl.edu,{jianghao,jarod.aerts}@huskers.unl.edu
†Department of Informatics, University of California, Irvine. malek@uci.edu

***Abstract*—In parallel with the increasing popularity of mobile software, an alarming escalation in the number and sophistication of security threats is observed on mobile platforms, remarkably Android as the dominant platform. Such mobile software, further, evolves incrementally, and especially so when being maintained after it has been deployed. Yet, most security analysis techniques lack the ability to efficiently respond to incremental system changes. Instead, every time the system changes, the entire security analysis has to be repeated from scratch, making it too expensive for practical use, given the frequency with which apps are updated, installed, and removed in such volatile environments as the Android ecosystem. To address this limitation, we present a novel technique, dubbed FLAIR, for efficient, yet formally precise, security analysis of Android apps in response to incremental system changes. Leveraging the fact that the changes are likely to impact only a small fraction of the prior analysis results, FLAIR recomputes the analysis only where required, thereby greatly improving analysis performance without sacrificing the soundness and completeness thereof. Our experimental results using numerous bundles of real-world apps corroborate that FLAIR can provide an order of magnitude speedup over prior techniques.**

***Index Terms*—Android analysis, evolving software, relational logic.**

I. Introduction

Android, with well over a million apps, has become the dominant operating system for mobile platforms [1]. The exponential growth in the popularity of the Android OS can be attributed in part to its flexible communication model, called inter-component communication (ICC), that facilitates sharing of data and services among applications. On the downside, the Android ICC interaction mechanism has become a vulnerable surface that causes serious security issues, such as privilege escalation chaining and app collusion, shown to be quite common in the Android apps on the market [2], [3], [4], [5], [6], [7]. Such issues are primarily due to the fact that the Android access control model is at the level of individual apps, and there is no mechanism to check the security posture of the entire system, 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.

Android security analysis, thus, has received a lot of attention in recent years to detect security issues that may occur due to the interaction of apps components [8], [9], [10], [2], [11], [12], [7], [13], [14], [15], [6], [16]. Pure program analysis techniques, such as IccTA [2], tried to address this problem through combining multiple apps, and treating all of the apps as one large program; this in turn enables performing program analysis on the entire system. Reliance on performing the analysis directly on a massive combined program, however, makes such techniques rather unscalable, as every time any of the apps changes, the entire analysis has to be repeated. Some hybrid approaches are more recently proposed that combine program analysis with other reasoning techniques [6], [17]. Among others, COVERT [6] combines static program analysis with bounded verification [18]. It shows that by decomposing the problem into two parts—(a) extracting security-relevant formal specifications from apps through program analysis, and (b) checking whether the extracted specifications are vulnerable or not through automated formal analysis—it is possible to become more effective, since the program analysis results do not need to be recomputed for the entire set of apps.

However, despite significant progress, all the prior approaches are subject to a common limitation: *they lack a way to respond to incremental system changes*, and vulnerability analysis has to be repeated from scratch for each system change, i.e., every time an app is added, removed or updated. Thus, security analysis of Android systems composed of a reasonable number of apps is still quite expensive to compute [2]. This is especially problematic in such volatile environments as the Android ecosystem, where apps are frequently being updated. Security analysis techniques based on a full recomputation of the analysis are often unscalable, and thereby, impractical in this context.

To address the foregoing challenge, this paper presents a novel technique, dubbed FLAIR for **f**ormally-precise evo**l**utionary **a**nalysis of **i**nteracting And**r**oid apps. It provides a novel Android-specific formal analyzer that automatically and efficiently updates ICC analysis results in response to incremental system changes. The insight underlying our research is that while apps comprising an Android system may evolve independent of one another, each change by itself is not likely to invalidate all the analyses that have been performed on an earlier version of the system, since not all apps in a system are capable of interacting with one another. Figure 1 presents a schematic view of FLAIR. Unlike prior techniques that dispose of all prior results in response to a system change, and redo the analysis from scratch, FLAIR employs an incremental technique for analyzing Android apps, where results of a prior

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00044

system analysis can be leveraged to optimize any subsequent security analyses on revisions of the system without sacrificing the soundness and completeness of the analysis.

We realize FLAIR on top of the open-source inter-component security analysis framework COVERT [6], where the Android system specifications are captured in Alloy relational logic [18]. Such specifications are amenable to fully automated, yet bounded, analysis. FLAIR replaces the standard Alloy relational logic analyzer [18], leveraged by COVERT as a backend analysis engine, with our new formal analyzer that automatically copes with incremental system modifications, without restricting the expressiveness and analyzability of the initial framework. The incremental analyzer relies on a set of algorithms that leverage the semantics of the change operations, i.e., adding/removing apps—that can be performed in the domain of the Android system—to narrow the search space of the revised system specification, thereby greatly reducing the required computational effort, further improving performance and scalability of the analyzer.

We evaluate FLAIR in the context of hundreds of real-world apps collected from a variety of repositories. Experimental results show that FLAIR significantly improves analysis efficiency, and that it is able to produce the same results as the state-of-the-art ICC analysis techniques based on a full recomputation, in significantly less amount of time.

To summarize, this paper makes the following contributions:

- *Incremental Android ICC analysis:* We introduce a novel approach to incremental analysis of Android ICC vulnerabilities. We present a set of algorithms for efficiently updating ICC analysis results based on the semantics of the change operations that can be performed in the domain of the Android system.
- *Tool implementation:* We implement FLAIR as an open-source extension to the Alloy relational logic analyzer and the COVERT inter-component analysis framework. We extend the latest version of the Alloy Analyzer to realize the incremental analysis algorithms, and incorporate it into the COVERT framework as a substitute for its back-end, non-incremental formal analysis engine. We make FLAIR publicly available to the research and education community [19].
- *Experimental evaluation:* We present our experiences with thorough evaluation of this approach in the context of real-world apps, the results of which corroborate that FLAIR can provide an order of magnitude speedup over conventional techniques.

The rest of this paper is organized as follows. Section II provides the background knowledge required to understand the contributions of our work. Section III presents the threat model. Section IV presents the core of this paper, FLAIR's algorithms for incremental analysis. Section V presents the evaluation of the research. Finally, the paper concludes with an outline of the related research and our future work.

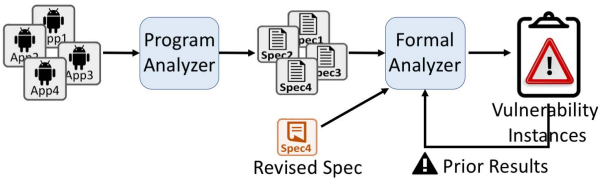

Fig. 1: A schematic view of FLAIR's incremental analysis approach that combines program analysis with lightweight formal analysis, where the formal analyzer recomputes the ICC analysis results in response to incremental system changes only where required.

II. BACKGROUND AND MOTIVATING EXAMPLE

This section provides a brief overview of the Android framework and its security challenges to help the reader follow the discussions that ensue, followed by an illustrative example to further motivate our research.

An Android system consists of a set of apps running on a device. Each app in Android consists of a set of software components. The most commonly used style of communication in Android is conducted by means of Intent messages [20]. 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 ICC allows for late run-time binding between components in the same or different apps, where the calls are not explicit in the code, rather made possible through event messaging, a key property of event-driven systems.

It has been shown that the Android ICC interaction mechanism introduces several security issues [4]. For example, Intent event messages exchanged among components can be intercepted or even tampered, since no encryption or authentication is typically applied upon them [21]. Moreover, no mechanism exists for preventing an ICC callee from misrepresenting the intentions of its caller to a third party [22]. In fact, the Android access control model provides no mechanism to patrol the security posture of the system as a whole. This causes several compositional security issues, such as Intent spoofing, unauthorized intent receipt, and privilege escalation, which are shown to be quite common in the apps on the market [4], [23], [24], [21], [25].

Figure 2 provides an example of such compositional security issues that we take as a running example in the rest of this paper. The first app is a *Messenger* app, where its *MessageSender* component uses system-level API `SmsManager`, resulting in a message to be sent to a phone number previously retrieved from an input Intent message. Although this app has the permission for SMS service, it fails to ensure that the sender of the original Intent message also has the permission. In fact, it defines a public interface, but fails to check if the caller has the proper permission. This represents a common practice, yet an anti-pattern, among Android developers [4]. The MessageSender component thus exposes the SMS service capability without checking the caller permission. The other

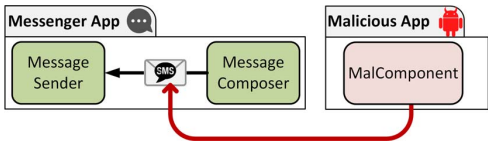

Fig. 2: A running example representing ICC vulnerabilities among Android apps.

app is a malicious app that sends an Intent message to MessageSender belonging to the vulnerable app for accessing the text message service, without the need for any permission.

Existing work on Android inter-component security analysis [9], [10], [2], [12], [7], [15], [6] can detect, within minutes to hours, that information from an Intent sent by a non-privileged app can flow to the MessageSender component. However, the current state-of-the-art lacks a way to respond to incremental system changes. Once an expensive analysis run of a system s has completed, whenever s changes, e.g. by installing a new app or even updating an existing one, the entire analysis will need to be recomputed thoroughly once again. This is especially problematic with a vast number of applications that nowadays are in use in a typical device. We show how through an effective incremental analysis, FLAIR can pragmatically address this challenge.

III. Threat Model

At the outset of any security analysis is the identification of a threat model that describes the capabilities of an attacker. This work is centered around efficient detection of three types of ICC-based security attacks in Android: (1) *Intent spoofing* [4], (2) *unauthorized intent receipt* [4], and (3) *privilege escalation* [23].

- **Intent spoofing** occurs when a malicious app forges an Intent to mislead a receiver app; essentially, a malicious app sends an Intent to an exported component of a victim app that is not expecting Intents from the malicious app.
- **Unauthorized Intent receipt** occurs when a malicious app intercepts an Intent meant for another legitimate app. As a result, the malicious app procures access to the data in the intercepted Intent, among other things.
- **Privilege escalation** occurs when an app with less privilege is not deprived of accessing components of a more privileged app. In fact, a malicious app is able to *indirectly* perform a privileged task, without having a permission to do so, by interacting with a component that possesses the permission.

We consider both explicit and implicit Intents as well as Intra- and Inter-apps communications. When the recipient component is given explicitly, the Intent is called an explicit Intent; otherwise, implicit Intent. We also consider the ICC-based communication that involves more than two apps, i.e., FLAIR will be able to detect inter-app attacks concealed in a *transitive ICC path* through multiple apps.

IV. Approach

This section first overviews the formal underpinnings of our approach based on relational logic as well as the COVERT framework, and then presents the details of our approach for incremental ICC analysis

A. Formal Underpinnings

Prior research has used Alloy's relational logic [18], and the corresponding Alloy Analyzer, for analysis of ICC vulnerabilities [6]. Our research builds on this work, but introduces novel enhancements to make the analysis significantly faster. Alloy is a declarative language based on the first-order relational logic with transitive closure [26]. The inclusion of transitive closure extends its expressiveness beyond first-order logic. Essential data types, that collectively define the vocabulary of a system, are specified in Alloy by their type signatures (sig). Signatures represent basic types of elements, and the relationships between them are captured by the the declarations of *field*s within the definition of each signature. Consider the following Alloy model. It defines two Alloy signatures: `Architecture` and `Component`. The `cmps` relation is defined over these two signatures.

```
sig Architecture{
  cmps: Component
}
sig Component{}
```

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

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

When executed, the Alloy Analyzer produces model instances, two of which are shown in Fig. 3. The model instance of Fig. 3a includes one architecture and two components, one of them belonging to no architecture. Fig. 3b shows another model instance with two architectures, each one having one component.

The other essential constructs of the Alloy language include: *Facts* and *Assertions*. A fact is a formula that takes no

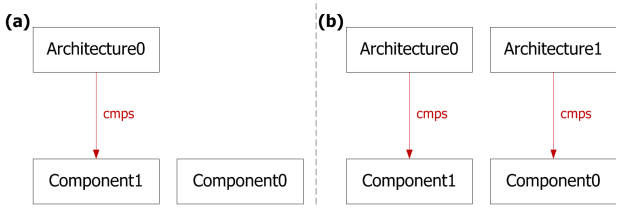

Fig. 3: Two model instances of the above Alloy specification.

arguments, and defines constraints that every instance of a model must satisfy, thus restricting the instance space of the model. An assertion (assert) is a formula required to be proved. It can be used to check a certain property of a model. The following fact paragraph, for example, states that each `Component` should belong to exactly one `Architecture`. Re-executing the Alloy Analyzer produces a new set of model instances, where while Fig. 3b is still a valid instance, model of Fig. 3a is eliminated.

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

We will introduce additional details of the Alloy language as necessary to present our incremental ICC analysis. For further information about Alloy, we refer the interested reader to [18].

B. COVERT *Overview*

COVERT is a formal analysis framework for automated, compositional analysis of Android apps [6]. It reduces the ICC vulnerability analysis problem into an Alloy relational logic problem. Through static analysis of application packages comprising a system, it automatically extracts architectural specifications annotated with security properties in the Alloy language in a way suitable for automated analysis. The Alloy Analyzer is then used to verify, albeit within a specific scope derived automatically for each system, whether it is safe for a combination of applications—holding certain permissions and potentially interacting with each other—to be installed simultaneously.

There are three main reasons that motivate our choice of COVERT as a platform for realizing the incremental ICC analysis in this work. First, its effective module system distinctly separates extraction of apps specifications from vulnerability analysis thereof. Such a well-structured module system that splits the overall, complicated system among tractable modules is not only convenient, but is an important part of our approach, as it enables effective compositional analysis of both model extraction and vulnerability analysis. Second, its reliance on a formal specification language, namely Alloy, and its associated analyzer facilitates orchestrating a cohesive implementation of the algorithms for updating Android inter-component analysis within a formally-precise analysis engine. Lastly, it is open-source and publicly available.

COVERT relies on two types of Alloy specifications: (1) Android framework specification, and (2) architectural specifications that it generates automatically for each Android app. The Android framework specification is a reusable specification, upon which all extracted architectural specifications are realized. It can be considered as an abstract specification that defines a set of rules to lay the foundation of Android apps, how they behave, and how they interact with each other. Listing 1 partially represents the *androidDeclaration* Alloy module, where the essential Android element types, such as Architecture, Component, etc., are defined as top-level Alloy signatures. These signatures are all specified as abstract, meaning that they cannot have an instance without explicitly extending them.

```
1  // Android framework specification
2  module androidDeclaration
3
4  abstract sig Application{
5    usesPermissions: set Permission,          //perms uses
6    appPermissions: set Permission          //perms enforces
7  }
8
9  abstract sig Component{
10   app: one Application,
11   iFilters: set IntentFilter,       //Component interfaces
12   permissions: set Permission,
13   paths: set Path                          //sensitive paths
14 }
15
16 abstract sig IntentFilter{
17   actions: some Action,                 //supported actions
18   data: set Data,
19   categories: set Category,
20 }
21
22 abstract sig Intent{
23   sender: one Component,
24   component: lone Component,          //recipient component
25   action: lone Action,
26   categories: set Category,
27   data: set Data,
28 }
29
30 abstract sig Path{
31   entry: one Resource,
32   destination: one Resource
33 }
34 abstract sig Permission{}
```

Listing 1: Excerpts from an Alloy specification of the Android application framework adapted from [6].

The `Application` signature contains two fields of `usesPermissions` and `appPermissions` that identify two sets of permissions (lines 4–7). The former declares the permissions to which the application needs to have access to run properly. The latter specifies the permissions required to access components of the application under consideration. `Components` are basic building blocks of Android applications, and the `app` field within the `Component` signature (line 10) identifies the parent application (architecture) to which a component belongs. Android applications can comprise four types of components, namely `Activity`, `Service`, `Receiver` and `Provider`. Signature declarations of four core component types extend the `Component` signature, omitted in the interest of space.

Component interfaces are specified as a set of `IntentFilters` that represent the kinds of requests a given component can respond to. A component may have any number of filters, captured by the `iFilters` field (line 11). The `permissions` field represents a set of permissions required to access a component. The `paths` then indicates information flows between sensitive resources, such as an ICC call method and a method that can trigger a permission-required operation (e.g., `sendTextMessage` in our running example).

The `IntentFilter` signature contains three fields of `actions`, `data` and `categories` (lines 16–20); the `actions` relation contains at least one element (due to the multiplicity keyword *some*), and `data` and `categories` map each `IntentFilter` instance to zero or more `Data` and `Category` objects, respectively.

```
1  // (a) Messenger app specification
2  module Messenger
3  open androidDeclaration
4  one sig Messenger extends Application{}{
5    usesPermissions = SEND_SMS
6    no appPermissions
7  }
8  one sig MessageSender extends Activity{}{
9    app in Messenger
10   iFilters = IntFilter1
11   no permissions
12   paths = path1
13 }
14 one sig path1 extends Path{}{
15   entry = ICC
16   destination = SMS
17 }
```

```
1  // (b) Malicious app specification
2  module MalApp
3  open androidDeclaration
4  one sig MalApp extends Application{}{
5    no usesPermissions
6    no appPermissions
7  }
8  one sig MalComponent extends Activity{}{
9    app in MalApp
10   iFilters = IntFilter2
11   no permissions
12   no paths
13 }
14 one sig intent1 extends Intent{}{
15   sender = MalComponent
16   component = MessageSender
17   action = SMS_SEND
18   no categories
19   data = Yes
20 }
```

Listing 2: Excerpts from automatically generated specifications for two illustrative apps shown in Figure 2.

The `Intent` signature contains five fields of `sender`, `component`, `action`, `data` and `categories` (lines 22–28). The first one denotes the component sending the Intent. The `component` field identifies the recipient component. As the keyword *lone* indicates an Intent may have either one or no declared recipient component, to capture whether the Intent is explicit or implicit, respectively.

The `Path` signature (lines 30–33) defines a path from each component's ICC `entry` point to an invocation of a permission-required functionality that is either inappropriately-guarded or unguarded, which may lead to ICC vulnerabilities. The last top-level signature is `Permission`. The COVERT framework captures both system-defined permissions and application-defined permissions, which are declared within the Android system's and the application's manifest files, respectively.

Listings 2a and b partially delineate the generated specifications for our running example apps (cf. Section II). The specifications start by importing the androidDeclaration module (line 3). The specification of Messenger app (Listings 2a, lines 4–7) then states that it has the permission for SMS service, yet does not enforce any permission that the other apps must have in order to interact with its components. The `MessageSender` component also contains a path from its ICC entry point to an invocation of the system-level `SmsManager` API, which enables sending a message to a phone number retrieved from the Intent.

The specification of Listings 2b shows that the MalApp does not declare any permission neither as required (`usesPermissions`) nor as enforced (`appPermissions`); yet has a Component of type Activity (line 8–20), which sends an explicit Intent to the `MessageSender` Component. This causes an inter-component permission leakage vulnerability, also called *privilege escalation*, where a component is able to make another component, here MessageSender, perform an action on its behalf, without having a proper permission.

To perform the compositional analysis on a set of formal models, COVERT includes specific Alloy signatures that model a set of security properties required to be checked. These signatures express properties that are expected to hold in the extracted specifications. The analysis is then conducted by exhaustive enumeration over a bounded scope of model instances to determine how the vulnerabilities and capabilities in individual apps could affect one another when the corresponding apps are installed together.

The combinatorial nature of the propositional formula to which the systems specifications are translated suppresses the size of systems for which an analysis can be performed within a reasonable amount of time. We believe exploiting an opportunity to take advantage of the problem domain, here the Android application framework (i.e., rules and constraints on the structure and behavior of its elements), to reduce the state space within which the analyzer explores would enable analyzing larger systems, that otherwise analysis of which is not possible. The following Section details our approach to incremental ICC vulnerability analysis.

C. Incremental ICC Analysis

Any change to a system, i.e., app addition/deletion, causes changes to its corresponding specification, which in turn, can render already analyzed vulnerability model instances stale. The non-incremental approach to this problem is to dispose of all the solutions, and recompute the analysis. The insight guiding our research is that the vulnerability analysis can be improved knowing that the changes in the particular domain of Android inter-component analysis are incremental, and often do not invalidate all of the solutions calculated in prior runs.

Specifically, each system change can be decomposed into a sequence of two operations of adding an app and removing an app. Note that an application update can be viewed as a remove operation followed by an add operation. The set of affected vulnerability instances caused by these two operations, however, is usually just a small fraction of all instances, offering a high potential for an incremental approach such as FLAIR. We start by demonstrating through Theorems 1 and 2 that the scope of changes being observed as a result of the app addition/deletion operations is limited to a small fraction of vulnerability instances computed in the prior analysis. We then describe how these theorems can effectively be realized in practice to enhance formal, yet incremental, analysis of ICC vulnerabilities.

Algorithm 1: Compute bounds for Add operation

Input: S^{prv}, S^{new} *//new and previous system specifications*
$I : S^{prv}.instances$ *//vul. instances for previous system spec.*
Output: $< alb, aub >$ *//adjusted lower and upper bound sets*

1 $< lb, ub > \leftarrow S^{prv}.bounds$
2 $< R^c, U^c > \leftarrow ComputeChanges(S^{prv}, S^{new})$
3 **for** $r \in S^{new}.relations$ **do**
4 **if** $r \notin R^c$ **then**
5 **if** $I \neq \emptyset$ **then**
6 $alb(r) \leftarrow \bigcap_{i \in I} i.val(r)$
7 $aub(r) \leftarrow ExtractAddedTuples(ub(r), U^c)$
8 $aub(r) \leftarrow aub(r) + \bigcup_{i \in I} i.val(r)$
9 **end**
10 **else**
11 $alb(r) \leftarrow lb(r)$
12 $aub(r) \leftarrow ExtractAddedTuples(ub(r), U^c)$
13 **end**
14 **end**
15 **else**
16 $aub(r) \leftarrow ub(r)$
17 $alb(r) \leftarrow lb(r)$
18 **end**
19 **end**
20 **return** $< alb, aub >$

Theorem 1. *Let a be an app not already installed on system S. Adding a to S does not eliminate any already existent ICC vulnerability, yet may cause new vulnerabilities.*

Proof. The proof of Theorem 1 is by contradiction: let us assume adding app a to system S eliminates an ICC vulnerability v already existent in S. Without loosing generality, we assume that there is a collection C of apps the interaction of which causes v. Because all apps involved in C are already installed on S before a is installed, vulnerability v is independent of a, i.e., regardless of app a being installed or not v is a valid vulnerability. This results in contradiction. Hence, our assumption that *adding app a to system S eliminates an already existent ICC vulnerability v* is false. □

Theorem 2. *Let a be an app already installed on system S. Removing a from S does not cause any new ICC vulnerability, yet may eliminate existent vulnerabilities, e.g., those caused by a.*

The proof of Theorem 2 is similar, thus omitted in the interest of space. Now, we can present FLAIR's algorithms for updating Android inter-component formal analysis in response to incremental system changes caused by adding and removing operations.

Specifications written in the Alloy language are first translated to bounded relational models in a language called Kodkod [27], which, in turn, are transformed into propositional formulas to be solved by SAT solvers. Kodkod allows specifying a scope over each relational variable from both above and below by two relational constants, called upper and lower bounds, respectively. The upper bound (*UB*) represents the whole set of tuples that a relational variable may contain, and a lower bound (*LB*) represents a partial solution for a given model. Every relation in a model instance, thus, must contain all tuples in the lower bound, and no tuple that is not in the upper bound.

The guiding principle in FLAIR is to adjust such bounds in evolutionary analysis of interacting apps, thereby limiting the scope of analysis that is examined in search of ICC vulnerability model instances.

D. Addition of a New App

Algorithm 1 presents how FLAIR computes bounds in response to Add operation. FLAIR first compares both system specification versions, computing a structural diff (Alg. 1, line 2). This diff gives information about which relations and tuples in the Kodkod's low-level bounded relational models were added or removed. Given the structural diff, one can express the differences between two system specification versions using two sets R^c and U^c, containing all changed relations and universe of elements (also called atoms), respectively. It then can infer from all changed tuples the set of affected vulnerability model instances, i.e, instance solutions which are outdated by the change and need to be updated. FLAIR treats as affected all vulnerability model instances to which the changed tuples contribute.

According to Theorem 1, the add operation does not eliminate any already existent ICC vulnerability instance, thus the set of lower bounds is set to include all determined vulnerability model instances; more specifically, for each relation the intersection of its values appeared in the model instances constitutes the lower bounds (Alg. 1, line 6). Note that while adding a new app might change the way Intents were previously delivered (e.g., the user may select the new app to handle certain type of requests, which in turn may prevent the MalApp from accessing the corresponding Intents), it would not eliminate the potential risk of the vulnerabilities already determined. Furthermore, given that the add operation may produce new ICC vulnerabilities, caused by the newly installed app, the set of upper bounds is set to include the union of all determined vulnerability model instances along with the tuples the new app introduced, allowing the analyzer to find potentially new ICC vulnerabilities (Alg. 1, lines 7–8). In case the vulnerability instance set, I, for the original specification is empty, FLAIR keeps the lower bound as initially calculated by the Alloy Analyzer; yet the upper bounds is set to include all the tuples affected by the newly added app, again allowing the analyzer to find potentially new ICC vulnerabilities. This is important in practice, since users are expected to install apps that are free of ICC vulnerabilities. Note that for the newly added relations, i.e., $r \in R^c$, FLAIR keeps the bounds unchanged (Alg. 1, lines 16–17).

This is a sound pruning of the model space, since elements of an added app, such as *Intents* and *Components*, etc., do not contribute to any value of the already found solutions, and the change has no effect on the present solutions for the evolving system specification. They, thus, constitute a partial solution for the updated system specification.

```
1 assert privEsc{
2   no disj src, dst: Component, i:Intent |
3     (src in i.sender) and
4     (dst in src.^transitiveICC) and
5     (some p: dst.app.usesPermissions |
6       not (p in src.app.usesPermissions) and
7       not ((p in dst.permissions) or
8            (p in dst.app.appPermissions)))
9 }
```

Listing 3: The assertion specification for privilege escalation in Alloy adapted from [6].

To make the idea concrete, consider the assertion specification for privilege escalation (Listing 3), one of the most prominent ICC vulnerabilities. The assertion, in essence, 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 by the victim component. Thus, it can be accessed by the src component through a chain of ICC calls. After analyzing the specification against our running example (cf. Section II), the following vulnerability model instance shown in Listing 4 is generated.

Note that the values assigned to the other relational variables in the vulnerability model instance are omitted in the interest of space. Now imagine that the user installs a new app (*App3*) that contains an Activity component (*App3Activity*). Let us see how FLAIR leverages the results of the previous run to set tighter bounds on relational variables. Among others, consider privEsc.src relation. Listing 5 shows its lower and upper bound sets, before (lines 1–3) and after (lines 5–7) being adjusted by FLAIR, in the Kodkod representation of our running example in its new setting.

Initially, the upper and lower bounds, calculated by the original Alloy Analyzer, contain three and zero elements, respectively. The upper bound, in fact, includes all component elements defined in the system specification. Out of these component elements, previous results show that *MalComponent* is actually a source of vulnerability, thus constitutes a partial solution for the updated system specification, and should be included in its lower bound (line 6). Also its union with the component newly added to the system constitutes the upper bound (line 7). As a result, for the particular relational variable of privEsc.src, the state space reduces from $2^{UB-LB} = 2^3 = 8$ possible values to $2^1 = 2$. This bound adjustment along with the similar adjustments applied to other relational variables would result in a considerable space reduction, which in turn improves the analysis time from 355 ms to 95 ms, in our simple running example.

```
1 ... // omitted details of model instances
2 privEsc.src = [MalApp/MalComponent]
3 privEsc.dst = [Messenger/MessageSender]
4 privEsc.i= [intent1]
5 privEsc.p= [appDeclaration/SEND_SMS]
```

Listing 4: An example vulnerability model instance for our running example.

Algorithm 2: Compute bounds for Remove operation

Input: S^{prv}, S^{new} *//new and previous system specifications*
$I : S^{prv}.instances$ *//vul. instances for previous system spec.*
Output: $< alb, aub >$ *//adjusted lower and upper bound sets*

1 $I \leftarrow S^{prv}.instances$
2 $< R^c, U^c > \leftarrow ComputeChanges(S^{prv}, S^{new})$
3 **for** $r \in S^{new}.relations$ **do**
4 $\quad aub(r) \leftarrow \bigcup_{i \in I}(i.val(r) - ExtractRemovedTuples(i.val(r), U^c))$
5 $\quad alb(r) \leftarrow \bigcap_{i \in I}(i.val(r) - ExtractRemovedTuples(i.val(r), U^c))$
6 **end**
7 **return** $< alb, aub >$

E. Removal of an Existent App

Algorithm 2 presents FLAIR's bounds computing algorithm in response to app removing operation. FLAIR again first compares both system specifications to compute changed relations and universe of elements in the Kodkod's low-level bounded relational models.

According to Theorem 2, the remove operation does not cause any new ICC vulnerability instance, but some existent vulnerabilities, and especially those caused by the removed app, are eliminated. The sets of upper and lower bounds, thus, are set to include all determined vulnerability model instances except those to which the removed app contributes, i.e., model instances in which values assigned to any of the relational variables include atoms from the removed app. As specified in Alg. 2 lines 4–5, for each relational variable the union and intersection of its values in all tuples, that are appeared in the vulnerability model instances but do not contain elements from the removed app, constitutes the upper and lower bounds, respectively.

As a concrete example, imagine that the user uninstalls the app that has just been added (*App3*). This time, the upper and lower bounds for privEsc.src, calculated by the original Alloy Analyzer, contain two (*MessageSender* and *MalComponent*) and zero elements, respectively. Previous results show that *MalComponent* is actually a source of vulnerability to which elements from the removed app do not contribute (cf. Listing 4). It, thus, constitutes a partial solution for the updated system specification. As a result, the state space for the privEsc.src variable is reduced from $2^2 = 4$ possible values to 1, making it a variable with exact bound (i.e., with an already known value), that does not need to be translated into a SAT formula, thus reducing the size of the generated SAT problem.

V. Experimental Evaluation

This section presents the experimental evaluation of FLAIR. We have implemented FLAIR as an open-source extension to the COVERT inter-component analysis framework and its backend Alloy analysis engine. To implement the algorithms presented in the previous sections, FLAIR modifies both the Alloy Analyzer and its underlying constraint solver, Kodkod [27]. Specifically, FLAIR's analyzer modifies the way in which

```
1 // The lower (lb) and upper (ub) bound sets for the privEsc.src relation before being adjusted.
2 lb: 0 []
3 ub: 3 [[privEsc, MessageSender], [privEsc, MalComponent], [privEsc, App3Activity]]
4
5 // The adjusted lower (alb) and upper (aub) bound sets for the same relation.
6 alb: 1 [[privEsc, MalComponent]]
7 aub: 2 [[privEsc, MalComponent], [privEsc, App3Activity]]
```

Listing 5: The lower and upper bound sets for the privEsc.src relation in the Kodkod representation of our running example, before and after being adjusted.

the Alloy Analyzer determines the scopes for each relational variable of the updated specification given the vulnerability model instances of the original system specification. The scope adjustments are then realized in the transformation of high-level Alloy specifications into low-level bounded relational models. Note that we also enhance the model extractor module of COVERT by leveraging two static analysis tools, namely FlowDroid [28] and IC3 [29]. FLAIR's tool and experimental data are available at the project website [19].

We used the FLAIR apparatus for carrying out the experiments. In our evaluation, we address the following research questions:

- **RQ1.** Does the approach enable incremental analysis of Android inter-component vulnerabilities in a manner consistent with a full recomputation of the analysis?
- **RQ2.** How well does FLAIR perform? What is the performance improvement achieved by FLAIR's incremental analysis compared to the state-of-the-art ICC analyzers?
- **RQ3.** How effective is our incremental analysis approach developed atop SAT solving technologies in reducing the size of transformed propositional formula? What is the overhead of FLAIR?

Experimental subjects. Our experimental subjects are a set of Android apps drawn from four repositories of Google Play [30], F-Droid [31], Bazaar [32] and MalGenome [33]. The Google Play store serves as the official Android app store, from which we collected the top 100 popular free apps. F-Droid is a software repository that contains free and open source Android apps. The collection of subject systems includes 300 apps from this repository. We also include 50 apps from Bazaar [32], a local app store, to cover the apps available in third-party repositories. Finally, it contains a collection of 50 malicious apps identified by the MalGenome project [33], a malware repository that covers the majority of existing Android malware families.

Experimental setup. To address the first research question, we use a suite of specifications developed for the security assessment of a set of Android apps in a prior work [6]. We compare the results of a full recomputation on the experimental subjects with the results of an incremental change analyzed using FLAIR.

To address the second research question, we measure and compare the analysis time taken by FLAIR with that of the state-of-the-art ICC analyzers, namely DidFail [9], DIALDROID [17], COVERT [6], SEALANT [7] and IccTA [2]. To perform the comparison experiments, we need to simulate configurations of apps installed on a device. To that end, we partition the set of apps under study into ten app bundles, each containing 50 apps randomly selected from the repositories. We choose this number of apps since it is higher than the average number of apps on a smartphone in the United States, which has shown to be approximately 41 [34]. These app bundles simulate collections of apps installed on end-user devices, and we use them to perform ten sets of independent experiments. We gradually increase the number of apps concurrently analyzed within bundles until each app bundle reaches to 50 apps. To show the performance of FLAIR when apps are removed from the system, we repeat the experiments while we gradually remove apps from each app bundle. Such bundles with gradually increasing/decreasing apps sizes provide us with a perfect suite of evolving systems that can be used for our experiments.

To address the last research question, we collect the number of variables and clauses in propositional formulas produced by both incremental and non-incremental techniques, i.e., FLAIR and COVERT, across experimental subjects. We also instrument FLAIR to measure its execution time while updating ICC analysis results in response to system changes. We used a PC with an Intel Core i7 2.4 GHz CPU processor and 16 GB of main memory, and leveraged Sat4J as the SAT solver during the experiments.

A. Results for RQ1 (Consistency)

To validate the consistency of results produced by FLAIR's incremental analysis with those produced by performing a full recomputation of the results, we applied FLAIR to the exact test cases that the COVERT project has been evaluated on, and the results of which are available online [35]. For two consecutive versions *v1* and *v2* of each app bundle (i.e., a set of Android apps deployed together), we first ran the COVERT analysis that uses the unmodified Alloy Analyzer on version *v2* and recorded the results. Afterwards, we ran FLAIR on *v1*, incrementally updated the results to version *v2*, and compared the results with that of the COVERT analyzer. Our experiments confirm that FLAIR computes the same results as a full recomputation in all cases, corroborating our theoretical expectation.

B. Results for RQ2 (Scalability)

We compared scalability of FLAIR with the other tools that support analysis of ICC vulnerabilities, namely IccTA [2], [5], DidFail [9], COVERT [6], DIALDROID [17], and SEALANT [7].

IccTA employs Epicc [15] and APKCOMBINER [36] to analyze multiple interacting apps [2]. It first merges all the

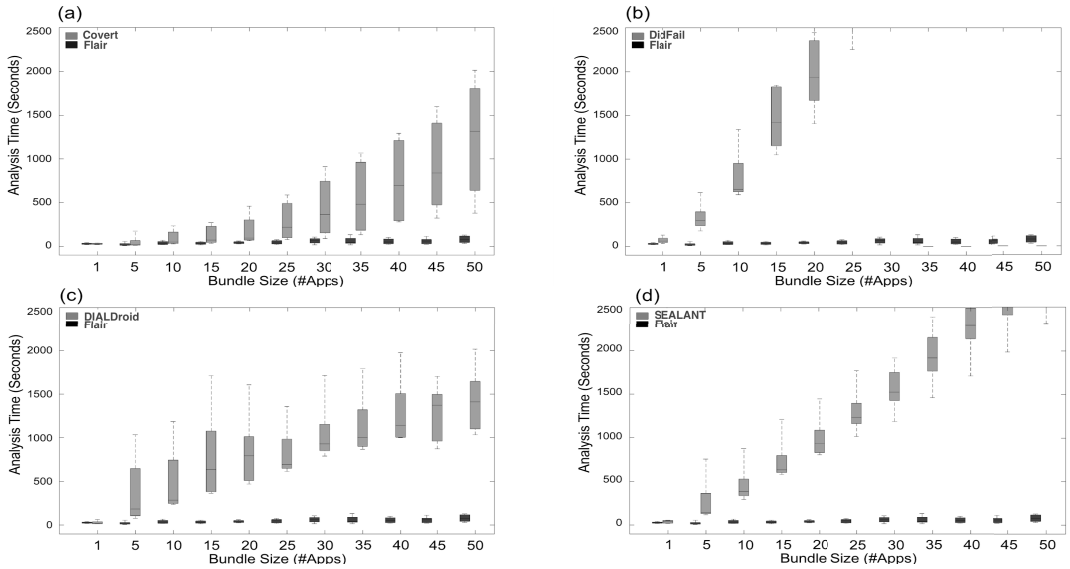

Fig. 4: The analysis time taken from (a) COVERT, (b) DidFail, (c) DIALDROID, and (d) SEALANT vs. FLAIR over the increasing size of the analyzed bundles as the number of apps under analysis increases. Note that analysis time taken by FLAIR tends to exhibit significantly lower growth rate than the corresponding time taken by the other state-of-the-art techniques.

apps with potential inter-app communication connections into a representation that resembles a single app, and then analyzes the new combined app by means of existing program analysis techniques to detect potential inter-app vulnerabilities. DidFail is another ICC analyzer [9], that was introduced at about the same time as IccTA and COVERT. It relies on FlowDroid [28] for taint analysis and Epicc [15] for ICC detection. DidFail first performs intra-app analyses on each single app and collects its manifest file as well as Epicc and FlowDroid outputs. Such intra-app analysis info captured from each individual app are then collectively analyzed to uncover vulnerable inter-app communications. DIALDROID is a program analysis tool aimed at identifying ICC-based vulnerabilities within large bundles of apps. Finally, SEALANT detects vulnerable ICC paths between apps through an integration of a data-flow analysis and a compositional ICC pattern matching.

The boxplots in Figure 4 show the analysis time taken by each of the techniques vs. FLAIR over the varying size of bundles, where the size of analyzed apps gradually increases. The number of apps is specified on the horizontal axis.

As illustrated in the diagram, the analysis time by DidFail scales exponentially, and for a bundle of 30 apps it exceeds one hour threshold. The analysis time by COVERT, DIALDROID and SEALANT grow significantly faster than the corresponding time for FLAIR. The effects of FLAIR's optimization are clearly visible when the size of app bundles increases. The median analysis time taken by FLAIR for a bundle of size 50 apps is 77 seconds with the interquartile range (IQR) of 67 seconds, corroborating that FLAIR can efficiently vet a large bundle of apps for ICC vulnerabilities.

Note that IccTA is not shown in the diagram because it was not able to analyze more than 2 apps in all our experiments. We noticed that the size of combined apps are indeed not increasing when the bundle size increases from 2 to 3. In fact, in most of our experiments, IccTA was unable to analyze more than one app at a time.

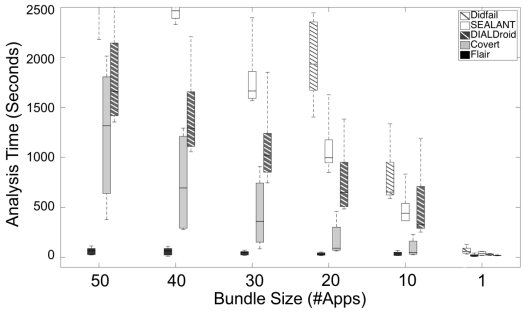

Fig. 5: The analysis time taken from each of the analysis techniques vs. the decreasing size of the analyzed bundles as the number of apps decreases.

The boxplot in Figure 5 shows the analysis time taken by each of the techniques over the decreasing size of the analyzed bundles as the number of apps gradually decreases. Each data point indicates the time it takes to reanalyze a revised system. Note that we remove apps gradually one by one, but for the sake of representation, the diagram shows box plots just for bundles whose sizes are multiples of ten. According to the diagram, the experimental data obtained from applying the remove operation give similar results as of the add operation. In summary, the results show that FLAIR outperforms the other state-of-the-art techniques in terms of scalability. FLAIR's incremental approach is able to analyze all bundles in a fraction of time that it takes from the others to analyze the same app bundles, and the difference in analysis time is more pronounced for the larger app bundles.

C. Results for RQ3 (Efficiency)

Table I shows the size of the propositional formulas generated by each of the two techniques, i.e., COVERT and FLAIR, across 10 subject app bundles, given as the total number of variables and clauses. Each bundle contains 50 apps. As shown, the number of variables and clauses in formulas generated by FLAIR is significantly less than those generated by COVERT. On average, FLAIR exhibits more than 92% reduction in the size of the translated propositional formulas, compared to those produced by COVERT that relies on standard Alloy Analyzer. This result clearly shows the effectiveness of our algorithm in reducing the exploration space.

The last column of Table I shows the FLAIR's adjustment time that pertains to the overhead incurred due to updating ICC analysis results in response to incremental system changes. According to the experimental data, FLAIR only introduces a small overhead (3.1% on average, and under 15% in all cases), but greatly outperforms the other state-of-the-art ICC analyzers, saving up to 94% of the analysis time without sacrificing vulnerability-finding ability.

VI. RELATED WORK

The work related to this paper falls into ICC analysis and incremental solving of constraints specified in first-order logic.

A large body of work focuses on Android ICC analysis [2], [12], [9], [6], [37], [38], [10]. Among others, IccTA leverages an intent resolution analysis to identify inter-component vulnerabilities [2]. IccTA's approach for ICC analysis is based on a pre-processing step connecting Android components through code instrumentation, which causes scalability issues. The main shortcoming of such purely program analysis techniques is that every time any of the apps changes, the entire analysis has to be repeated. This paper addresses this shortcoming by separating model extraction from actual ICC analysis, and by providing a novel Android-specific formal analyzer that automatically and efficiently updates ICC analysis results in response to incremental system changes.

The other relevant thrust of research has focused on incremental solving of constraints specified in first-order logic [39], [40], [41], [42]. Among others, Titanium extends the Alloy Analyzer to support analysis of evolving specifications [42]. This research effort shares with ours the emphasis on improving the analysis performance. Our work differs fundamentally in its emphasis on developing an incremental, relational logic analyzer for the particular domain of Android ICC analysis. FLAIR, thus, takes an efficient, domain-specific approach for narrowing the state space of all relational variables, instead of a general approach based on declarative slicing to identify a number of relational variables whose bounds can be adjusted.

Along the same line, Uzuncaova and Khurshid partitioned a model of constraints into a base and derived slices, where solutions to the base model can be extended to generate a solution for the entire model [40]. The problem that they addressed is, however, different form ours. They tried to leverage model decomposition to improve scalability. Whereas, given a system specification that is already analyzed, FLAIR updates the analysis results in response to incremental changes. Ranger [43] uses a divide and conquer method relying on a linear ordering of the solution space to enable parallel analysis of specifications written in first-order logic. While the linear ordering allows for partitioning of the solution space into ranges, there is no clear way in which it can be used for incremental analysis of evolving systems.

TABLE I: The size of variables and clauses in propositional formulas produced by COVERT and FLAIR for app bundles of size 50 apps along with the analysis overhead.

	COVERT		FLAIR		
	Vars	Clauses	Vars	Clauses	Adjustment Time (Sec)
B.1	361,854	61,752,270	1,230	10,212	1.721
B.2	367,893	63,408,733	7,331	1,389,184	1.812
B.3	279,683	40,513,124	7,448	617,554	1.563
B.4	358,141	53,563,717	8,334	762,302	2.442
B.5	351,145	52,646,280	1,282	10,544	4.806
B.6	318,195	48,644,466	21,517	1,679,991	4.148
B.7	273,416	39,750,850	1,122	9,095	2.987
B.8	407,094	53,941,151	1,399	11,407	15.028
B.9	414,615	55,447,247	8,970	1,268,853	4.026
B.10	297,890	44,525,013	1,171	9,482	33.052
Avg.	342,993	51,419,285	5,980	576,862	7.15

VII. CONCLUSION

This paper presents FLAIR, a novel approach for efficient and incremental security analysis of evolving Android systems. FLAIR's update algorithm is based on reducing the space of values to be explored by the SAT-solver underlying the analysis engine. We have implemented FLAIR on top of Alloy, its underlying relational logic analyzer, Kodkod, and the COVERT inter-component analysis framework. The experimental results of evaluating FLAIR in the context of hundreds of real-world Android apps corroborates its ability to provide an order of magnitude speedup over the state-of-the-art, non-incremental analysis techniques.

While our focus in this paper was on ICC analysis, we believe such an effective exploration space reduction for the bounded analysis of relational logic can pave the way for application of formal analyses in a wide range of problems in, among others, software design [44], [45], [46], [47], [48], [49], code analysis [50], [51], and test case generation [52], [53]. Our future work will explore the application of techniques described in this paper (e.g., analysis bound adjustment) to other software engineering problems.

ACKNOWLEDGEMENT

We thank Alireza Sadeghi for his help with the COVERT framework and helpful feedback on an early draft of the paper. This work was supported in part by an NSF EPSCoR FIRST award, and awards CCF-1755890, CCF-1252644, CNS-1629771 and CCF-1618132 from the National Science Foundation, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

REFERENCES

[1] R. Cozza, I. Durand, and A. Gupta, "Market Share: Ultramobiles by Region, OS and Form Factor, 4Q13 and 2013," *Gartner Market Research Report*, February 2014.

[2] L. Li, A. Bartel, T. Bissyande, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE 2015, Florence, Italy, 2015.

[3] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. Raleigh, NC: ACM, 2012, pp. 229–240.

[4] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. Washington, DC: ACM, 2011, pp. 239–252.

[5] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis," *arXiv:1404.7431 [cs]*, Apr. 2014, arXiv: 1404.7431. [Online]. Available: http://arxiv.org/abs/1404.7431

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

[7] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A *SEALANT* for inter-app security holes in android," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 312–323.

[8] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Trans. Software Eng.*, vol. 43, no. 6, pp. 492–530, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2615307

[9] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and 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*. Edinburgh, UK: ACM, 2014, pp. 1–6.

[10] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. Scottsdale, AZ: ACM, 2014, pp. 1329–1341.

[11] B. R. Schmerl, J. Gennari, A. Sadeghi, H. Bagheri, S. Malek, J. Cámara, and D. Garlan, "Architecture modeling and analysis of security in android systems," in *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, ser. Lecture Notes in Computer Science, B. Tekinerdogan, U. Zdun, and M. A. Babar, Eds., vol. 9839, 2016, pp. 274–290. [Online]. Available: https://doi.org/10.1007/978-3-319-48992-6_21

[12] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, "Multi-app security analysis with FUSE: Statically detecting android app collusion," in *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, ser. PPREW-4. New Orleans, LA: ACM, 2014, pp. 4:1–4:10.

[13] M. Hammad, H. Bagheri, and S. Malek, "Determination and enforcement of least-privilege architecture in android," in *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*. IEEE, 2017, pp. 59–68. [Online]. Available: https://doi.org/10.1109/ICSA.2017.18

[14] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek, "A temporal permission analysis and enforcement framework for android," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE'18, 2018, pp. 846–857.

[15] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. USENIX Association, 2013, pp. 543–558.

[16] D. Octeau, S. Jha, M. Dering, P. D. McDaniel, A. Bartel, L. Li, J. Klein, and Y. L. Traon, "Combining static analysis with probabilistic models to enable market-scale android inter-component analysis," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 469–484. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837661

[17] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, 2017, pp. 71–85.

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

[19] "Flair web page," https://sites.google.com/view/flairanalysis/, 2018.

[20] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic, "Software architectural principles in contemporary mobile software: from conception to practice," *Journal of Systems and Software*, vol. 119, pp. 31–44, 2016. [Online]. Available: https://doi.org/10.1016/j.jss.2016.05.039

[21] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *13th International Conference*, ser. ISC'10, M. Burmester, G. Tsudik, S. Magliveras, and I. Ili, Eds. Boca Raton, FL, USA: Springer Berlin Heidelberg, Oct. 2010, pp. 346–360.

[22] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "QUIRE: Lightweight provenance for smart phone operating systems." in *USENIX Security Symposium*, San Francisco, CA, 2011.

[23] S. Bugiel, L. David, Dmitrienko, T. A. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.

[24] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Technische Universitt Darmstadt, Technical Report TR-2011-04*, 2011.

[25] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. Chicago, IL: ACM, 2011, pp. 627–638.

[26] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[27] E. Torlak, "A constraint solver for software engineering: Finding models and cores of large relational specifications," PhD Thesis, MIT, Feb. 2009. [Online]. Available: http://alloy.mit.edu/kodkod/

[28] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, ser. PLDI'14. Edinburgh, UK: ACM, 2014, p. 29.

[29] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Int'l Conf. on Software Engineering*. Florence, Italy: IEEE, May 2015.

[30] "Google play market," http://play.google.com/store/apps/, 2017.

[31] "F-droid," https://f-droid.org/, 2017.

[32] "Bazaar," 2017. [Online]. Available: https://cafebazaar.ir//.

[33] "Malgenome project," http://www.malgenomeproject.org, 2017.

[34] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti, "Predicting user traits from a snapshot of apps installed on a smartphone," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 18, no. 2, pp. 1–8, 2014.

[35] "Alloy models from the covert project," http://www.sdalab.com/projects/covert, 2015.

[36] L. Li, A. Bartel, T. F. Bissyand, J. Klein, and Y. L. Traon, "ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis," in *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, ser. ICT SEC'15, H. Federrath and D. Gollmann, Eds., vol. 455. Springer, 2015, pp. 513–527.

[37] H. Bagheri, A. Sadeghi, R. J. Behrouz, and S. Malek, "Practical, formal synthesis and automatic enforcement of security policies for android," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 2016, pp. 514–525. [Online]. Available: https://doi.org/10.1109/DSN.2016.53

[38] A. Sadeghi, H. Bagheri, and S. Malek, "Analysis of android inter-app security vulnerabilities using COVERT," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 725–728. [Online]. Available: https://doi.org/10.1109/ICSE.2015.233

[39] S. Ganov, S. Khurshid, and D. E. Perry, "Annotations for alloy: Automated incremental analysis using domain specific solvers," in *Proc. of ICFEM*, 2012, pp. 414–429.

[40] E. Uzuncaova and S. Khurshid, "Constraint prioritization for efficient analysis of declarative models," in *Proc. of International Symposium on Formal Methods*, ser. FM'08, 2008.

[41] ——, "Kato: A program slicing tool for declarative specifications," in *Proc. of International Conference on Software Engineering*, ser. ICSE'07, 2007, pp. 767–770.

[42] H. Bagheri and S. Malek, "Titanium: Efficient analysis of evolving alloy specifications," in *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. FSE'16, 2016.

[43] N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias, "Ranger: Parallel analysis of alloy models by range partitioning," in *Proceeding of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 147–157.

[44] H. Bagheri and K. J. Sullivan, "Model-driven synthesis of formally precise, stylized software architectures," *Formal Asp. Comput.*, vol. 28, no. 3, pp. 441–467, 2016. [Online]. Available: https://doi.org/10.1007/s00165-016-0360-8

[45] H. Bagheri, C. Tang, and K. J. Sullivan, "Trademaker: automated dynamic analysis of synthesized tradespaces," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 106–116. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568291

[46] ——, "Automated synthesis and dynamic analysis of tradeoff spaces for object-relational mapping," *IEEE Trans. Software Eng.*, vol. 43, no. 2, pp. 145–163, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2587646

[47] H. Bagheri and K. J. Sullivan, "Bottom-up model-driven development," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 1221–1224. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606683

[48] ——, "Pol: specification-driven synthesis of architectural code frameworks for platform-based applications," in *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, K. Ostermann and W. Binder, Eds. ACM, 2012, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/2371401.2371416

[49] H. Bagheri, Y. Song, and K. J. Sullivan, "Architectural style as an independent variable," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 159–162. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859026

[50] J. P. Near and D. Jackson, "Derailer: Interactive security analysis for web applications," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 587–598. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643012

[51] M. Taghdiri, "Inferring specifications to detect errors in code," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 144–153. [Online]. Available: http://dx.doi.org/10.1109/ASE.2004.42

[52] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. Williams, Eds. ACM, 2016, pp. 559–570. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884853

[53] S. Khurshid and D. Marinov, "Testera: Specification-based testing of java programs using SAT," *Autom. Softw. Eng.*, vol. 11, no. 4, pp. 403–434, 2004.