Received: 30 September 2021 | Revised: 24 October 2022 | Accepted: 25 October 2022

DOI: 10.1002/spe.3171

RESEARCH ARTICLE

WILEY

A model-based framework for inter-app Vulnerability analysis of Android applications

Atefeh Nirumand[1] | **Bahman Zamani**[1] | **Behrouz Tork-Ladani**[1] | **Jacques Klein**[2] | **Tegawendé F. Bissyandé**[2]

[1]MDSE Research Group, Department of Software Engineering, University of Isfahan, Isfahan, Iran

[2]Interdisciplinary Center for Security, Reliability and Trust, University of Luxembourg, Esch-sur-Alzette, Luxembourg

Correspondence
Bahman Zamani, Model-Driven Software Engineering Research Group, Department of Software Engineering, University of Isfahan, Isfahan, Iran.
Email: zamani@eng.ui.ac.ir

Abstract

Android users install various apps, such as banking apps, on their smart devices dealing with user-sensitive information. The Android framework, via Inter-Component Communication (ICC) mechanism, ensures that app components (inside the same app or on different apps) can communicate. The literature works have shown that this mechanism can cause security issues, such as app security policy violations, especially in the case of Inter-App Communication (IAC). Despite the plethora of research on detecting security issues in IAC, detection techniques face fundamental ICC challenges for improving the precision of static analysis. Challenges include providing comprehensive and scalable modeling of app specification, capturing all potential ICC paths, and enabling more effective IAC analysis. To overcome such challenges, in this paper, we propose a framework called VAnDroid2, as an extension of our previous work, to address the security issues in multiple components at both intra- and inter-app analysis levels. VAnDroid2, based on Model-Driven Reverse Engineering, has extended our previous work as per following: (1) providing a comprehensive Intermediate Representation (IR) of the app which supports extracting all the ICC information from the app, (2) extracting high-level representations of the apps and their interactions by omitting the details that are not relevant to inter-app security analysis, and (3) enabling more effective IAC security analysis. This framework is implemented as an Eclipse-based tool. The results of evaluating VAnDroid2 w.r.t. correctness, scalability, and run-time performance, and comparing with state-of-the-art analysis tools well indicate that VAnDroid2 is a promising framework in the field of Android inter-app security analysis.

KEYWORDS

android, inter-app communication, inter-component communication, model-driven reverse engineering, security analysis

1 | INTRODUCTION

Nowadays, smartphones have become an integral part of users' daily life, because users are constantly installing various apps, such as banking and medical apps, on their devices. It is estimated that more than 2 billion Android apps have

been developed so far.[1-3] As these apps can deal with user-sensitive information, the insecurity of apps will have serious consequences on users' lives.[2] When developing an app, there are different aspects of security to be considered. Therefore, there is a basic need for Android app developers, app distributors, and app analysts to have easy-to-use automated techniques and tools for the security analysis of apps, including vulnerability analysis.[1]

Three types of program analysis are mostly used to check the security properties of Android applications: static, dynamic, or hybrid.[4,5] In static analysis, security checks are performed without the actual execution of the program code. Compared with dynamic analysis, static analysis is often more scalable and covers more execution paths of the program.[6] Therefore, this paper is focused on static analysis. In this context, several studies[7-11] have been conducted to identify various security issues of Android applications, such as vulnerabilities and information leaks. However, several challenges can be enumerated for the Android static analysis methods. Following are the five most recurrent challenges.

(1) Since Android apps are event-driven, and app components can interact with each other via the Inter-Component Communication (ICC) mechanism, mainly through Intent messages, the execution paths in these apps are unpredictable, hence, it is challenging to identify all ICC paths. As Android apps become more complex software, many potential execution paths must be considered in the analysis.[12] Therefore, one of the significant challenges in the static analysis of Android apps is capturing and analyzing all possible ICC paths to avoid losing potentially dangerous behavior while not introducing too many false alarms.[11]
(2) According to recent studies,[1,6,13] most of the current static analysis approaches suffer from low precision. Furthermore, it should be noted that identifying more security issues in Android applications highly depends on the ability of analytical methods to characterize the app specification accurately.[2] In this characterization, the entire Android specification related to the Android app structure must be considered. Therefore, there is a fundamental need for studies to improve the precision of static analysis techniques in characterizing the app specification. These techniques must be applicable to large and complex apps.[5]
(3) The Android framework ensures that the apps produced by a wide variety of developers are able to interact when installed on a single mobile phone, as long as these interactions comply with the restrictions imposed by this framework.[5] While Google provides several best practices, many Android app developers fail to properly follow these practices for Android secure programming, allowing malicious apps designed to misuse the IAC mechanism to trick vulnerable Android apps into performing activities beyond their privilege. Investigating these features (i.e., the Android communication model) will reveal a new set of security issues.[5] The main reason for these issues is that the Android access control model operates at the level of individual apps and does not provide any mechanism to check the security status of the entire system (Inter-app communication [IAC] level).[14] However, according to the literature review studies,[6,13,15] most of the existing research are limited to the analysis of a single app. While today security issues such as vulnerabilities and information leaks through the incorrect implementation of the IAC mechanism are ubiquitous,[2] it is necessary to identify security problems in communication between apps.
(4) Fragmentation, evolution, and upgrading of the Android Application Programming Interfaces (APIs) often block the existing ICC analysis tools.[12,16] Therefore, when developing new approaches, researchers need to be able to adapt their work to new versions of Android APIs.
(5) In unstable environments like Android, where apps are constantly added, removed, or updated, there is a fundamental need to conduct more scalable and practical ICC analysis, such as incremental ICC analysis that uses the results of previous analyses to optimize subsequent analyses and automatically updates ICC analysis to respond to the app changes.[14]

Given the above challenges, the field of security analysis of Android applications demands an automated framework for providing comprehensive and scalable modeling to enable more effective security analyses. Such a framework can be used to address the security issues in IAC.

One way to cope with the structural complexity of software systems is to create a high-level abstraction (i.e., characterization) of these systems and focus only on the specifications required for analysis.[17] Model-Driven Reverse Engineering (MDRE) is one of the approaches that benefit from this idea and uses models to decrease the heterogeneity of systems. We argue that using MDRE solutions in the field of security analysis of Android apps could be helpful and has several advantages, including extensibility, full coverage, (re)use, and integration.[17,18] Our earlier work[19] proposed a framework called VAnDroid, which stands for "Vulnerability Analysis of Android Applications." This framework takes advantage of MDRE to perform a more effective intracomponent analysis of Android apps. VAnDroid[19] is only able to detect the security issues in a single app component.

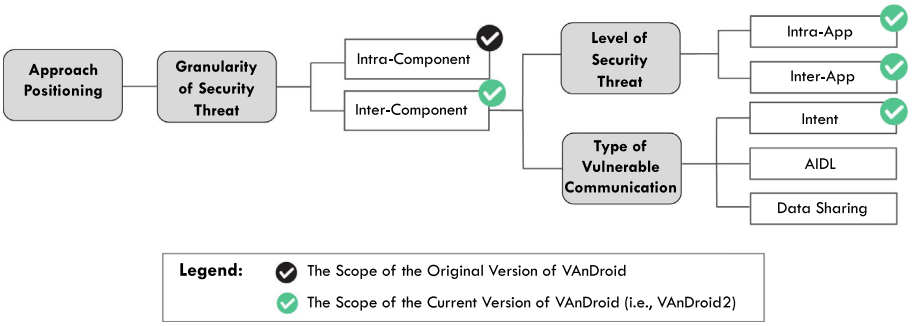

FIGURE 1 The positioning of the original VAnDroid framework[19] and the VAnDroid2 framework (based on the taxonomy proposed by Sadeghi et al.[13])

To overcome the challenges mentioned above, in this paper, an automated model-based framework called VAnDroid2 is proposed. This framework is an extension of the original VAnDroid framework[19] to add the ability to take advantage of MDRE to conduct a more effective intercomponent analysis and improve the detection of security issues at intra- and inter-app levels. In this framework, technology-independent standards (i.e., metamodels) are used to extract models that have the required features for Android app security analysis. It is also possible to determine precisely how these features relate to security issues, such as the types of vulnerabilities. Furthermore, it is possible to enable inter-app security analysis by paying a one-time cost of generating model-based representations from Android applications.

To clearly specify the boundary of the contribution as well as the extensions and enhancements made to the original VAnDroid framework,[19] we use Figure 1, inspired by the taxonomy provided by Sadeghi et al.[13] for Android security analysis. As depicted in this figure, the original VAnDroid framework[19] is intended to address security issues in a single app component (i.e., intracomponent analysis). VAnDroid2 made substantial extensions to the original VAnDroid framework[19] for addressing the security issues in multiple app components (i.e., intercomponent analysis). Also, VAnDroid2 is intended to consider both a single app (i.e., intra-app analysis) and a combination of apps (i.e., inter-app analysis). Therefore, in VAnDroid2, four levels of analysis are supported: intracomponent (single component), intercomponent (multiple components), intra-app (single app), and inter-app (multiple apps). Among the three major types of interprocess communication (IPC) mechanisms (i.e., Intent, Android Interface Definition Language [AIDL], and data sharing), VAnDroid2 focuses on Intent.

The VAnDroid2 framework has been developed to generate comprehensive and scalable models of the Android app specification to enable more effective IAC security analysis. At the heart of this framework is a model-based static analysis approach for Android applications, implemented to enable an incremental and automated analysis of the security specification and structure of Android apps that are constantly being installed, removed, and updated on user's devices. The VAnDroid2 framework consists of three phases. In the Model Discovery phase, through model-based static analysis, the comprehensive Intermediate Representation (IR) of each app in a bundle is created without losing information. Then, in the transformation and Integration phase, by collecting security information in the form of domain-specific models from each app, the comprehension of the Android system is facilitated and all potential inter-component communication (inside the same app or among different apps) are captured from a bundle of apps in an analyzable domain-specific model. Finally, in the analysis phase, according to the resulting models, a formal analysis process is conducted to support both intra- and inter-app vulnerability analysis.

The proposed framework has been developed as an Eclipse-based tool. It can be used to identify two prominent inter-app vulnerabilities called Intent Spoofing and Unauthorized Intent Receipt. To evaluate the VAnDroid2 tool, it has been applied to 10 bundles of real-world Android apps to examine the criteria of correctness, scalability, and run-time performance. These bundles, each containing 35 apps, have been randomly selected from the provided dataset in this study (i.e., a collection of benign, malicious, and vulnerable Android applications). The VAnDroid2 tool is also compared with several existing state-of-the-art tools related to ICC analysis of Android apps: IC3,[20] IccTA,[9] Amandroid,[11] and COVERT.[10] The evaluation results indicate that VAnDroid2 has conducted a more effective IAC security analysis and achieved higher precision and recall in intra- and inter-app vulnerability detection. Research artifacts, including the tool and evaluation results, are available on the VAnDroid2 website*.

*https://mdse.ui.ac.ir/project/vandroid2/

To summarize, this paper makes the following contributions (significant extensions from the original VAnDroid framework[19]).

(1) **Model-based static program analysis for Android**. We presented a model-based static program analysis for Android, which supports extracting all the ICC information from an Android app needed for analyzing ICC at intra- and inter-app levels.
(2) **Model-based ICC extractor for Android**. We developed a model-based ICC extractor for Android to precisely infer all potential ICCs at intra- and inter-app communication levels. This ICC extractor implements a model-based app component analysis that conducts a precise intent resolution (matching) algorithm to extract all potential ICCs from a bundle of Android apps.
(3) **A formal model-based analysis process**. We implemented model-to-model (M2M) transformations through ATL (as a model transformation language) and OCL (as a formal language) rules, which supports the identification of two prominent ICC vulnerabilities at intra- and inter-app communication levels.
(4) **An extended version of the Android application security aspects metamodel**. The metamodel is proposed to extract the security information from each app in a bundle of apps and integrate them into a single Android Application Security Aspects model. This model is reusable and detailed enough that it can be conducted to perform a more effective inter-app security analysis. This metamodel is an extension of the metamodel defined in our earlier work.[19] This metamodel is extensively enhanced to consider the more complete specification of Android apps, including various types of app components, all kinds of intent objects, and enforced permissions by components, as a set of permissions required to access components of an app. We also improved the implementations for extracting the Android Application Security Aspects model to perform an elementary string analysis. In this analysis, intent parameters are extracted to identify the more precise correspondence between the source and targets for ICC. The details of this metamodel are described in Section 5.1.
(5) **A metamodel for ICC of Android apps**. This brand new metamodel is proposed to extract all potential ICCs in a bundle of apps at the intra- or inter-app levels and integrate them into a single model. In order to extract this model, a model-based algorithm has been implemented to perform app components analysis. This algorithm conducts the precise Intent resolution process, which makes it possible to support both intra- and inter-app analysis.
(6) **Implementation of the proposed framework as an Eclipse-based tool**. To show the ability of the proposed approach to perform both intra- and inter-app vulnerability analysis, it is developed as an incremental and automated Eclipse-based tool called VAnDroid2. This tool receives multiple apps (app bundles) and performs incremental ICCs analysis to identify security issues at four analysis levels: intracomponent, intercomponent, intra-app, and inter-app. VAnDroid2 is extendable, and all components of this tool can be used for other types of inter-app vulnerabilities.
(7) **Experiments**. The evaluation results of the VAnDroid2 tool w.r.t. correctness, scalability, and run-time performance have been presented. These experimental evaluation results corroborated VAnDroid2's ability to perform effective inter-app ICC analysis. VAnDroid2 is also compared with several state-of-the-art tools (i.e., IC3, IccTA, Amandroid, and COVERT) related to ICC and IAC analysis of Android applications. The results of the comparisons indicate that VAnDroid2 outperforms the state-of-the-art tools. In comparison with IC3, as a program analysis tool for Android, VAnDroid2 significantly outperforms the IC3 tool both in extracting more comprehensive specifications from each app and execution time. In comparison with IccTA and Amandroid, as two intra-app analysis tools for Android, VAnDroid2 outperforms the other two tools and achieves higher precision (100%), recall (96%), and F-measure (98%) in intra-app ICC analysis. In comparison with COVERT, as an inter-app analysis tool for Android, VAnDroid2 significantly outperforms COVERT and achieves higher precision (100%), recall (100%), and F-measure (100%) in inter-app ICC analysis.

The remainder of this paper is organized as follows. Section 2 presents background information required to understand this study. Section 3 reviews the related work. Section 4 explains an illustrative inter-app vulnerability example to motivate our research. Section 5 introduces an overview of the proposed approach, the proposed metamodels, and tool support. The evaluation results are presented in Section 6. We discuss evaluation results, limitations, and threats to validity in Section 7. Finally, Section 8 concludes the paper and highlights the areas of future work.

2 | BACKGROUND

This section provides background information on Android apps, ICC, access control model, and MDRE concepts.

2.1 | Android applications

As the most popular mobile platform, Android accounted for 73% market share in June 2021.[21] This platform includes the Linux operating system, middleware, system libraries, and a set of pre-installed Android applications.[5] The Android platform architecture consists of several layers. At the bottom of this platform, there is a hardware layer that contains the hardware components of the Android system. The Linux layer, as the second layer, is the core of the Android System. This layer is responsible for the proper operations of all system components.[22] At the top of this layer, there is a Hardware Abstraction Layer (HAL). This layer contains several library modules that provide interfaces for each hardware component. Finally, at the top of the Android system, there is an application layer that provides the API for Android apps to interact with the system. These applications can be pre-installed apps that exist on the Android system by default or installed by users from various repositories, which are called third-party apps.[22]

Android apps are distributed as the Android Package (APK) file formats.[4] An APK consists of several files. The Manifest file is a mandatory and important file for any app which contains metadata about the app, including the app components, required permissions, and enforced permissions. Components are the primary and logical units of the Android app, which can be following four types:[4]

- Activity. It is the basis of the application interface. An Android app may have several activities that display different pages of the app to the user.
- Service. It provides the background processing of the app, such as playing music. This component does not provide any user interface.
- Broadcast Receiver. It responds to the system's broadcast messages. The Receiver also operates as a gateway for other app components and sends messages to handle services or activities.
- Content Provider. It provides the ability to share data between Android applications.

2.2 | Intercomponent communication

Android separates apps from each other and system resources using the sandbox mechanism as part of the protection mechanism. This separation requires interaction through a message passing system called ICC.[5] Figure 2 illustrates this procedure and shows that the components of two Android apps, *Maps* and *Facebook*, can communicate with each other through the ICC mechanism. As depicted in this figure, app components can interact with each other inside the same app or among different apps.

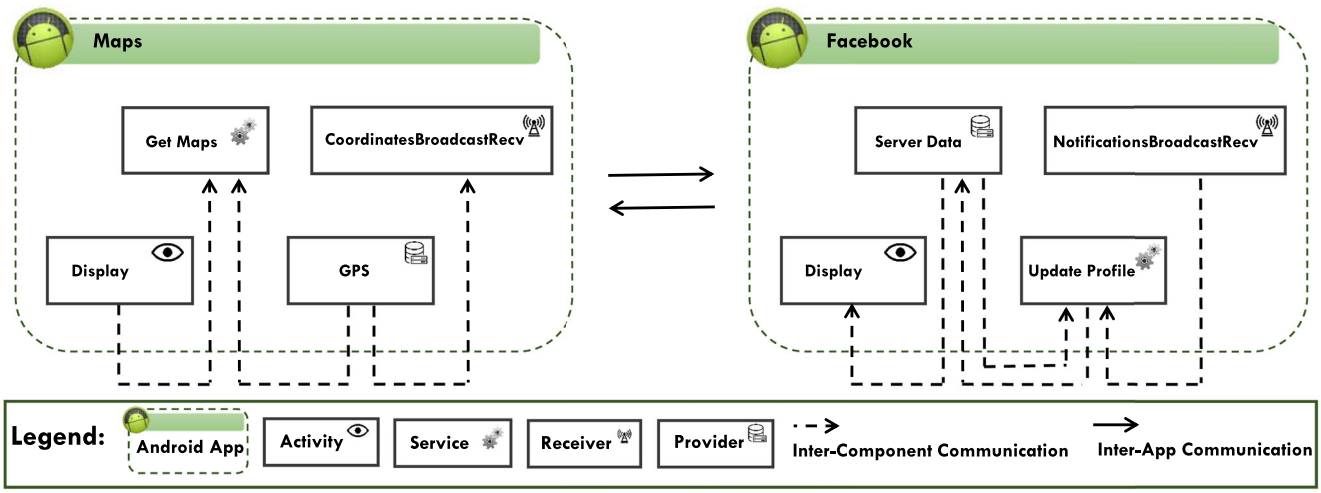

FIGURE 2 An example of app component interactions through the intercomponent communication mechanism (adapted from Reference 15)

The ICC mechanism is done through Intent messages or Uniform Resource Identifiers (URIs).[4] Intent objects provide an abstract description of the app's capabilities. On the other hand, the component's capabilities are determined by a set of intent filters specified in the Manifest file.[10] An Intent message specifies an event to perform a specific action with data that supports that action.[4] The components of Activity, Service, and Receiver can communicate with each other via Intent messages. URIs are used to communicate with Content Provider, as a database for an Android app.[4] An Intent message can be addressed to a specific app component, which is called an explicit intent. However, an Intent message can be sent implicitly, which is called an implicit intent. In this type of intent, the component receiving the intent is not explicitly specified.[15]

In the ICC mechanism, components can be invocated through Intent messages at two levels: intra-app (inside the same app) or inter-app (among different apps).[4] As shown in Table 1, the Android framework provides several ICC methods that can be used for sending intent messages. In order to identify these ICC methods, the works of Chin et al.,[23] Ma et al.,[24] Samhi et al.,[25] and Android API reference documentations[26-28] have been studied. App components can be activated by calling these intent-sending mechanisms like startActivity, startService, and sendBroadcast. Also, some methods are related to sending Intent through PendingIntent and IntentSender objects.[5,25] PendingIntent, as a wrapper around the Intent, allows the Intent's action to be performed in the future, even when the original sender app of intent is not active anymore.

TABLE 1 A nonexhaustive list of intercomponent communication methods[23-28]

Usage	Method
To receiver	sendBroadcast(intent i)
	sendBroadcast(intent i, String rcvrPermission)
	sendOrderedBroadcast(intent i, String recvrPermission, BroadcastReceiver receiver, …)
	sendOrderedBroadcast(intent i, String recvrPermission)
	sendStickyBroadcast (Intent i)
	sendStickyBroadcast(intent i)
	sendStickyOrderedBroadcast(intent i, BroadcastReceiver receiver, …)
To activity	startActivity(intent i)
	startActivityForResult(intent i, int requestCode)
	onActivityResult(int requestCode, int resultCode, intent intent)
	setResult(int resultCode, intent intent)
To service	startService(intent i)
	bindService(intent i, ServiceConnection conn, int flags)
PendingIntent	send(Context context, int code, Intent intent, PendingIntent.OnFinished onFinished, …)
	send(int code, PendingIntent.OnFinished onFinished, Handler handler)
	send(Context context, int code, Intent intent)
	setExact(int type, long triggerAtMillis, PendingIntent operation)
	requestLocationUpdates(long minTimeMs, float minDistanceM, Criteria criteria, PendingIntent pendingIntent)
	requestLocationUpdates(String provider, long minTimeMs, float minDistanceM, PendingIntent pendingIntent)
	getActivity(Context, int, Intent, int)
	getActivities(Context, int, Intent[], int)
	getBroadcast(Context, int, Intent, int)
	getService(Context, int, Intent, int)
	getIntentSender()
IntentSender	sendIntent(Context context, int code, Intent intent, IntentSender.OnFinished onFinished, Handler handler)
	sendIntent(Context context, int code, Intent intent, IntentSender.OnFinished onFinished, Handler handler, String requiredPermission)

2.3 | Android access control model

Android access control model is implemented at the level of individual Android applications. In this access model, two types of privileges are considered for app components:[4] (1) *ICC* that allows a component to interact with other app components in the same or among different apps. (2) *Resource access privilege* that allows an app component to access resources of the mobile device. As explained in Section 2.2, ICC is done through Intent messages or URIs. Since the ICC mechanism is mainly done through Intent messages, this paper is focused on the message passing mechanism through intent.

While various mechanisms are considered in the Android access control model, including the Android permission model, this access model does not provide any mechanism to check the security status of the entire system and its applications.[14,29] Therefore, several malicious apps can combine their permissions or take advantage of benign apps' vulnerabilities to perform activities beyond their privilege. This causes important security issues in inter-app communications, including Intent Spoofing and Unauthorized Intent Receipt attacks,[5,23] described in the following.

2.3.1 | Intent spoofing

In this security attack, the malicious app component can communicate with a public (exported) app component while not expecting such communication. An app component is public if the exported attribute for this component is set to true or if it has at least one Intent Filter. If the victim (exported) app component blindly trusts the incoming (received) intent, the malicious app component can cause the victim component to perform malicious actions.[4,23] This security attack can be three types according to the type of victim app component.[4,23] These three types are described as follows.

- Malicious activity launch. This attack occurs when an exported activity of the victim app is initiated by a malicious app component that does not expect such communication. Since the activity component provides the GUI interface, this attack can be used to deceive the user. If this attack is successful, it can modify the background data according to the Intent's data sent by the malicious component. Also, information leakage may occur via the victim Activity.
- Malicious Service launch: This attack occurs when a malicious app component can start an exported Service of the victim app. This type of attack is similar to a malicious activity launch attack, except that the interaction between the victim and the malicious components occurs in the background. If this attack is successful, the victim service may leak sensitive information or perform unauthorized tasks.
- Malicious broadcast injection. This attack occurs when a public (exported) receiver blindly trusts the intent sent by the malicious component. Since the Broadcast Receiver operates as a gateway for other app components and sends messages to handle services or activities, the malicious intent can propagate throughout the application. As a result, this attack can perform an inappropriate operation on the data from the intent or even run operations that the malicious component is not supposed to trigger.

2.3.2 | Unauthorized intent receipt

In this security attack, the malicious app component can intercept an implicit intent by introducing a filter that matches the sent intent. Therefore, the malicious app component can access all the data on the intent.[4,23] This security attack can be three types according to the type of malicious component.[4,23] These three types are described as follows.

- Activity hijacking. This attack occurs when a malicious Activity component has received the intent. As a result, this Activity is launched instead of the legitimate Activity.
- Service hijacking. This attack occurs when a malicious Service is initiated instead of a legitimate Service. If this attack is successful, it may trigger a false response attack in which the malicious result is returned to the Intent sender.
- Broadcast theft. This attack occurs when the receiver app component can silently read the content of the broadcast intent without any interruption in the Receiver component.

2.4 | Model-driven reverse engineering

As a paradigm for software engineering, model-driven engineering (MDE) focuses on creating, manipulating, and using models.[17] These models can describe various complementary aspects of software systems. In MDE, models are considered as first-class entities in the design, development, deployment, maintenance, and evolution of software systems. Therefore, many benefits can be achieved by moving from code-based approaches to model-based ones, including increasing the level of abstraction, increasing problem understanding to better complexity management, and improving the overall efficiency of various software engineering tasks.[17]

MDE is based on three main concepts: model, metamodel, and model transformation. The metamodel determines the possible element types and relationship types of the model that conform to it. Model transformation can be done as model-to-model transformation (e.g., Eclipse ATL[30]) or model-to-text transformation (e.g., Eclipse Acceleo[31]). The first type specifies a mapping from the source metamodel to the target metamodel. The second type, also called code generation, specifies a mapping from the source metamodel to the grammar of the target language.[17,32,33]

MDRE takes advantage of the MDE principles and techniques in reverse engineering to develop more effective solutions that facilitate the understanding of software systems.[18] MDRE obtains a set of models from the software system artifacts (e.g., configuration files and source code).[33] MDRE uses models to reduce the structural complexity of software systems. The heterogeneity of the systems decreases with the use of homogeneous models. MDRE can also be directly exploited from MDE technologies and their capabilities, such as generality, extensibility, integration, and coverage.[34] An MDRE process consists of three phases as follows:[33]

- Model Discovery. High-level representations (models) are obtained from software artifacts.
- Model Understanding. The models obtained in the previous phase are understood.
- Model (Re)generation. The generated models in the previous phase can be used to produce new models or to migrate the software system into a new platform.

3 | RELATED WORK

There exist extensive research work on Android security analysis in general and on security vulnerability detection in particular. In the following, an overview of the researches that are most closely related to our research is given. First, in Section 3.1, an overview of the studies related to performing program Analysis of Android apps for security is provided. Then, in Section 3.2, the studies related to performing ICC analysis of Android apps for vulnerability detection are described.

3.1 | Program analysis of Android apps for security

Over the past decade, significant studies have been made to improve the security of Android applications. These studies have resulted in the development of tools for analyzing Android applications. In the following, an overview of the tools in light of our research is explained.

CHEX[35] is a static analysis tool that automatically analyzes a single Android app to detect component hijacking vulnerability. CHEX identifies this type of vulnerability through data flow modeling. This modeling is done by analyzing the Android application to identify the data stream being hijacked. CHEX is based on Dalysis, as a framework proposed by Lu et al.[35] for performing various static analyses of Android bytecode.

Epicc[7] is an Android static analysis tool that detects security issues related to the Android communication model and analyzes the app components that can interact with each other. This tool converts the Dalvik bytecode to Java bytecode via Dare[36] and then creates an ICC call graph for Android app analysis. The approach used in this tool is flow-sensitive, inter-procedural, and content-sensitive.

FlowDroid[8] is an Android taint analysis tool. This tool implements an analysis that is object, context, field, and flow-sensitive. FlowDroid models the complete lifecycle of an Android app. This tool evaluates the configuration files and bytecode to detect existing privacy leaks.

DroidSafe[37] examines the information flow in the Android application code and detects possible leaks of sensitive information. This tool well models the Android execution environment and provides a combination of an accurate, comprehensive, and precise model of the Android execution environment with design decisions.

IIFA,[38] as a static analysis tool, performs information flow analysis of Android apps to identify potential data leaks. This tool examines sensitive information flows that can occur through intent messages. These information flows are not limited to one component and can occur between the components in the same or different apps.

3.2 | ICC analysis of Android apps for vulnerability detection

Since our work focuses on ICC analysis to detect inter-app vulnerabilities, in the following, an overview of some of the researches that considered ICC analysis and vulnerability detection is described.

Chin et al.[23] studied the Android communication model to identify ICC attacks that can occur through intent messages. They also developed the ComDroid tool to detect two types of vulnerabilities called Intent Spoofing and Unauthorized Intent Reception through static analysis of Android applications. Android app developers can conduct this tool to analyze their apps before release.

DidFail[39] performs a static analysis of Android applications. This tool combines Epicc[7] and FlowDroid[8] analyses to accurately track data flow in the app bundles at intercomponent and intracomponent levels. DidFail identifies data flows in each app separately. It then identifies potentially dangerous flows that may occur.

IccTA[9] is an Android static analysis tool for performing ICC analysis. This tool identifies privacy leaks between components by constructing a highly precise control-flow graph. In this tool, the APK file is received as input, and the bytecode of the Android application is transformed into Jimple, as a Soot's [†] internal representation. Then, the ICC links are extracted, and based on these links, Soot-based representations are modified to analyze the data flow between components. It also extracts a complete control-flow graph from the whole Android application.

Jha et al.[40] proposed a conceptual model to display ICC in Android apps at a higher abstraction level from the code. They also developed an automated tool called ICCMATT to perform ICC modeling and testing of Android apps. This tool extracts an ICC graph from the source code to specify intra- and ICCs within the Android app. Android app developers and analysts can use this call graph to identify ICC vulnerabilities.

COVERT[10] has been developed to address IAC, in which a combination of static analysis with formal methods is applied. The main purpose of this tool is to detect permission-induced issues in Android apps, namely permission-induced security attacks and permission-induced compatibility defects.

Amandroid[11] is a tool that addresses the security problems such as data leaks, data injection, and API abuse. Initially, this tool converts the app's Dalvik bytecode to an IR. Amandroid creates an environment model that emulates communications of the Android system with the app. This environment is similar to the environment created by the FlowDroid,[8] but unlike an app-level environment model used in FlowDroid, it is a component-level environment model. Amandroid performs the component-based analysis. For this purpose, for each app component, a Data Flow Graph (DFG) is generated, and the component-level data dependence graph (DDG) is created based on the DFG. In fact, this tool creates an abstraction of the app behavior using a DFG and a DDG at the component level.

Based on the review of related studies, we realized that (1) most of the solutions are proposed to address the detection of sensitive information leaks in Android apps, (2) the majority of related studies are limited to consider ICC analysis at the intra-app level (i.e., a single Android app), and (3) Most static analysis methods have used Soot framework, as an IR of the code. We will later, in Section 7.2 (Discussion), compare the proposed approach with the related works.

4 | MOTIVATING EXAMPLE

In this section, we illustrate an inter-app vulnerability where a malicious app component performs an inter-app Intent Spoofing attack. In this example, a bundle from the Ghera repository[1] is considered. The Broadcast Receiver component in Android apps can be dynamically registered in the Java code (i.e., at runtime).[14] This type of component is automatically considered as an exported component without any access restriction. Therefore, any app can access this component via the ICC mechanism, even malicious apps. As a result, unintended operations may be triggered by this component.[1]

[†] https://github.com/soot-oss

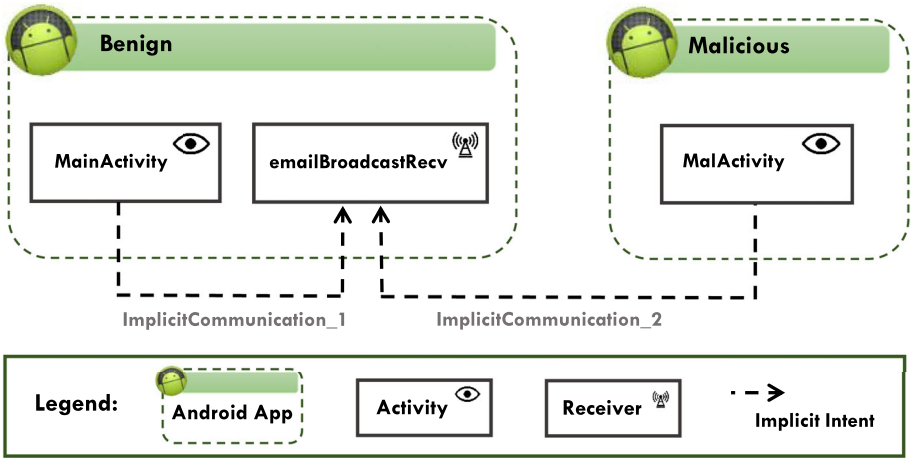

FIGURE 3 A running example of inter-App vulnerabilities

Consider the bundle shown in Figure 3. This bundle contains a benign app and a malicious app (i.e., *Benign* and *Malicious*). *Benign* has a component called *MainActivity*. The code excerpt of this component is shown in Listing 1. As can be seen, in line 3, there is an implicit intent with an action *edu.ksu.cs.action.EMAIL*. *Benign* registers a Broadcast Receiver that can dynamically receive the intent. The Java code snippet related to registering this Broadcast Receiver is shown in Listing 2. This Broadcast Receiver has an intent filter with an action *edu.ksu.cs.action.EMAIL* (line 6) and can be triggered by any app via the intent, even a malicious app.

```
1 @Override
2 public void onClick (View v) {
3 Intent intent = new Intent ("edu.ksu.cs.action.EMAIL");
4 intent.putExtra("email", "diagnostics@startup.com");
5 intent.putExtra("text", "I am " + username);
6 sendBroadcast(intent);
7  }
```

Listing 1: Code excerpt of MainActivity of *Benign*

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3  super.onCreate(savedInstanceState);
4  setContentView(R.layout.activity_main);
5  emailBroadcastRecv = new EmailBroadcastRecv();
6  registerReceiver(emailBroadcastRecv, new IntentFilter("edu.ksu.cs.action.EMAIL"));
7  }
```

Listing 2: Code excerpt of register the dynamic Broadcast Receiver in *Benign*

```
1 @Override
2   public void onClick(View v) {
3   Intent intent = new Intent("edu.ksu.cs.action.EMAIL");
4   intent.putExtra("email", "rookie@malicious.com");
5   intent.putExtra("text", "I can send email without any permissions");
6   sendBroadcast(intent);
7  }
```

Listing 3: Code excerpt of MalActivity of *Malicious*

The second app is a malicious app. This app has an Activity called *MalActivity*. This component has a filter with an action *android.intent.action.MAIN*. The code excerpt of *MalActivity* is shown in Listing 3. In line 3, this component has an implicit intent with an *action edu.ksu.cs.action.EMAIL*. *Malicious* can invoke the Broadcast Receiver in *Benign* and exploit it to send the email, although *Malicious* is not allowed to send an email.

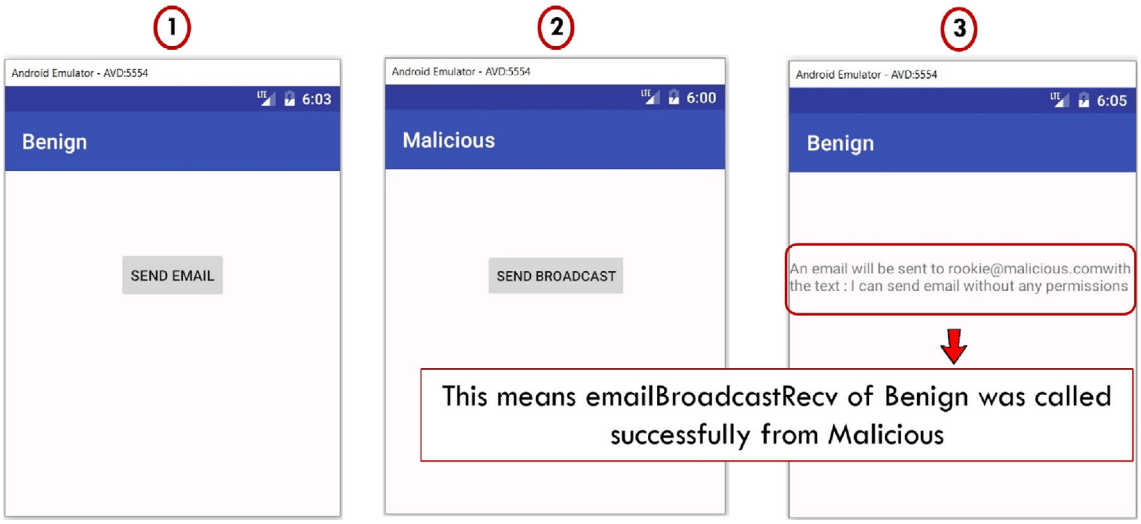

FIGURE 4 The attack scenario for *Benign*

The scenario of *Malicious* attacking *Benign* is shown in Figure 4. As can be seen, after running *Malicious*, *Benign* receives the intent from *Malicious* and then sends it through an email, and the message specified by *Malicious* is displayed. While sending emails is not very dangerous, the Broadcast Receiver registered dynamically can very well be exploited by a malicious app component to perform more dangerous operations. This is a common practice (i.e., anti-pattern) among the developers of Android applications.[23]

The above example is one of the most important vulnerabilities in IAC called Intent Spoofing, which we consider as an example of an inter-app vulnerability throughout this paper.

5 | THE PROPOSED APPROACH

This section overviews the proposed approach to enable a more effective ICC analysis and detection of security issues at both intra- and inter-app levels. As explained earlier, this approach addresses two prominent ICC vulnerabilities called Intent Spoofing and Unauthorized Intent Receipt. The proposed approach starts with receiving a bundle of Android apps. The comprehensive IR of each app is first created. Then, the security information from each app is extracted and integrated into a single domain-specific model. By collecting this model from each app, the comprehension of the Android system is facilitated and all potential ICCs (inside the same app or among different apps) are extracted in the form of an analyzable domain-specific model. Finally, two prominent ICC vulnerabilities, Intent Spoofing and Unauthorized Intent Receipt, are identified by performing formal analysis operations (i.e., queries and model manipulation techniques).

To extract the security information from each app, the android security aspects metamodel proposed in our earlier work[19] has been enriched and extended to consider the more complete specification of Android apps. Also, the ICC metamodel is proposed to capture all potential ICCs from the Android system.

In the following, first, the proposed metamodels are explained in Sections 5.1 and 5.2. Then, in Section 5.3, an overview of the proposed approach and its phases are presented. Finally, the incremental ICC analysis feature and the tool support are explained in Sections 5.4 and 5.5, respectively.

5.1 | Android application security aspects metamodel

For automatic security analysis of IAC, a model of each app is required to identify its security structure and specification, which conforms to the metamodel shown in Figure 5. This metamodel is an extension of the proposed metamodel by our earlier work.[19] The elements with dark colors indicate the extended concepts. In the following, Sections 5.1.1-5.1.8 describe the main elements of this metamodel and their relationships.

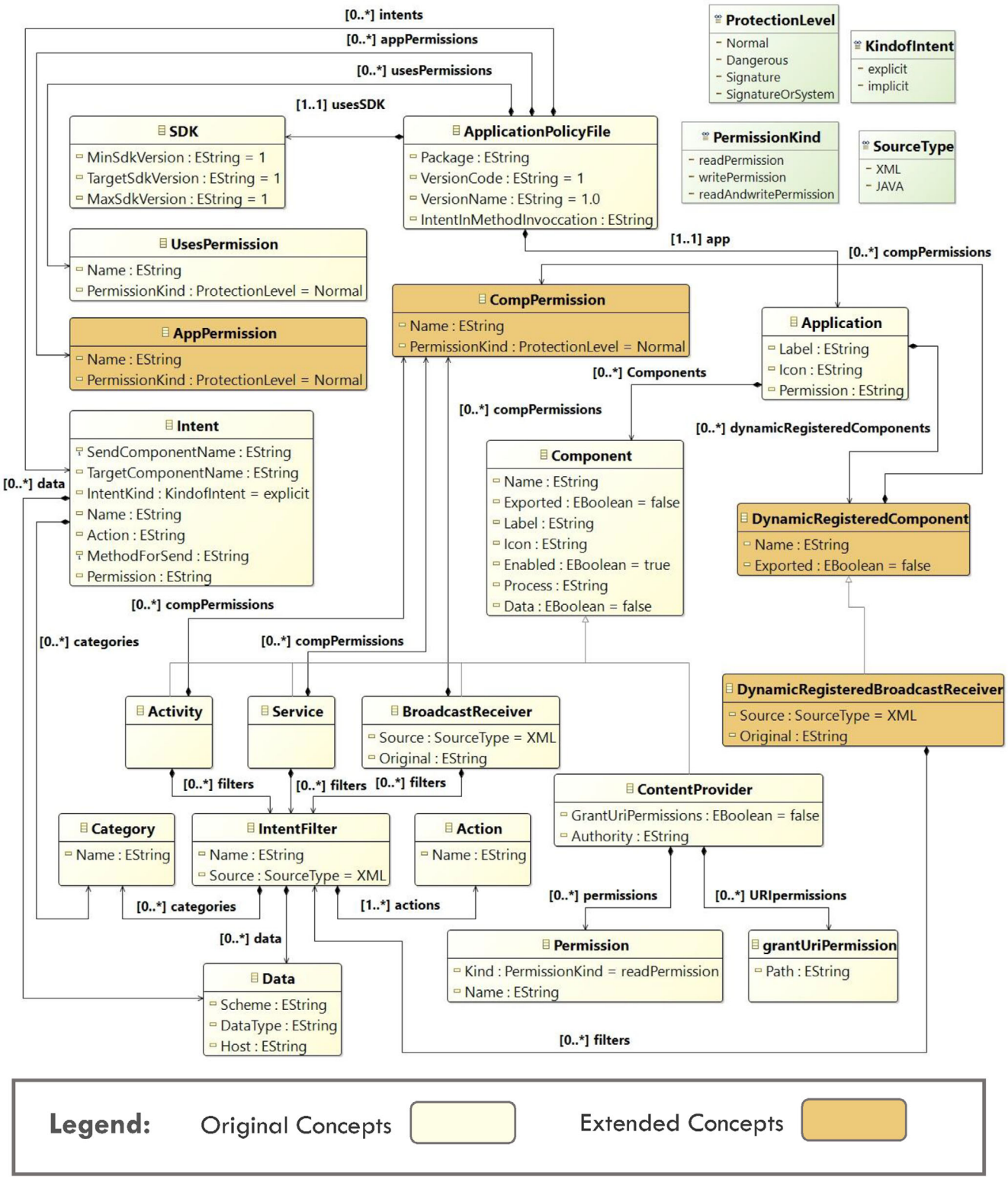

FIGURE 5 Android application security aspects metamodel (extended from the proposed metamodel by our earlier work,[19] the extended concepts are in dark color)

5.1.1 | ApplicationPolicyFile

This element represents the root of the metamodel, which contains essential information of the Android application: (1) *Package* that identifies the name of APK file, (2) *VersionCode* that is needed for other apps to identify the version of the app, and 3) *VersionName* that specifies the version of the app shown to users.

5.1.2 | SDK

The *SDK* element identifies the Android API level required to run the application. This element contains three attributes: (1) *MinSdkVersion* that specifies the Android API minimum level required to run the application, (2) *MaxSdkVersion* that specifies the Android API maximum level required to run the application, and (3) *TargetSdkVersion* that specifies the Android API level, which the Android application is designed to run on it.

5.1.3 | UsesPermission

Android conducts a permission-based mechanism that restricts the access of Android apps to critical resources, other apps, or other app components.[41] These requested permissions by the app are stored in the application's sandbox and granted to all app components.[42] This model of coarse-grained Android permissions violates the principle of least privilege.[43] Hence, malicious apps may exploit this mechanism for activities beyond their privilege.[41] Therefore, the information related to requested permissions is one of the necessary security aspects of Android apps for static ICC analysis.

The *UsesPermission* element in the metamodel represents the requested permissions concept. This element specifies a set of permissions that an application needs to access protected parts of the system or other apps, both system-defined and application-defined permissions.

This element contains two attributes of *Name* and *Permissionkind* that identify the name and the type of requested permission, respectively. Android permissions are divided into four levels: normal, dangerous, Signature, and SignatureOrSystem, which are considered through *ProtectLevel* in the metamodel.

5.1.4 | AppPermission

This element is a set of permissions required to access components of an Android application, which contains two attributes of *Name* and *Permissionkind* that identify the name and the type of required permission, respectively.

5.1.5 | Component

As explained before, an Android app can contain four types of components: Activity, Service, Receiver, and Content Provider. The *Component* element represents this concept of the Android app. All app components must be statically introduced in the Manifest file, but the Receiver component can also be dynamically introduced in the Java code.[14] This type of component is considered through *DynamicRegisteredComponent* in the metamodel.

5.1.6 | IntentFilter

This element represents a set of intent filters specified for a component. Activity, Service, and Receiver can have a set of intent filters, each of them specifies a different capability of the component. Intent filters for Broadcast Receivers registered dynamically must be declared in their Java code (i.e., at runtime), which is specified through *SourceType* in the metamodel.

5.1.7 | CompPermission

If an app component is public (exported), the other app components (inside the same app or on different apps) can access this app component. However, a component can specify permissions to restrict access. The *CompPermission* element represents a set of permissions for a component that other components must have to communicate this component. This element contains two attributes of *Name* and *Permissionkind* that identify the name and the type of required permission, respectively.

5.1.8 | Intent

As discussed in Section 2.2, app components can interact with each other mainly through Intent messages. The *Intent* element represents a set of intent messages that can be used for intra- and inter-app communication. This element has the following attributes:

- Name. This attribute specifies the name of intent.
- Action. Each intent can contain at most one action, which represents the general action that must be performed by the app component receiving the intent.
- Permission. It indicates the required permission to limit the number of app components that can receive the broadcast intent.
- IntentKind. It specifies the type of intent that can be of two types: explicit and implicit.
- SendComponentName. This attribute specifies the name of the component that creates the intent.
- TargetComponentName. This attribute specifies the name of the component that should receive and handle the intent (this attribute for explicit intent must be explicitly specified).
- MethodForSend. It indicates the used ICC methods for intent-sending mechanisms.

Each intent also has two sets: (1) *Data*, which specifies additional information related to the data that must be processed by the specified action. (2) *Category*, the intent object can also have a set of categories.

5.2 | ICC metamodel

Regarding the Android access control model (described in Section 2.3), this research focuses on addressing ICC at the intra- and inter-app levels. As discussed earlier, ICC in Android is mainly implemented through the use of intent messages. Each of these intent messages is a specific event that executes an action on data that supports that action. The capabilities of the components are then determined by a set of filters that indicate the type of requests that the component can handle. In fact, intent filters provide interfaces for a component. Components can be invocated in different ways: (1) explicit, (2) implicit, (3) intra-app, or (4) inter-apps.[43] To extract potential communications in the same or different apps via intent messages, the metamodel shown in Figure 6 is presented. This metamodel is described in the following.

InterComponentCommunicationInBundleOfApps is the root of the metamodel, which contains two sets that indicate two communication domains, as follows.

- *Explicit Communication* represents all potential interactions that can be made through explicit intent. This domain is considered as *ExplicitCommunication* in the metamodel.
- *Implicit Communication* represents all the potential interactions that can be made through implicit intent. This domain is considered as *ImplicitCommunication* in the metamodel.

SenderApp indicates the Android application whose component creates the intent. *ReceiverApp* specifies an Android application that can receive the intent.

ExplicitIntent represents all explicit intent objects that exist in a bundle of Android apps. This element contains three attributes: (1) *Name*, the name of intent, (2) *SendComponentName*, the sender component name of the intent, and (3)

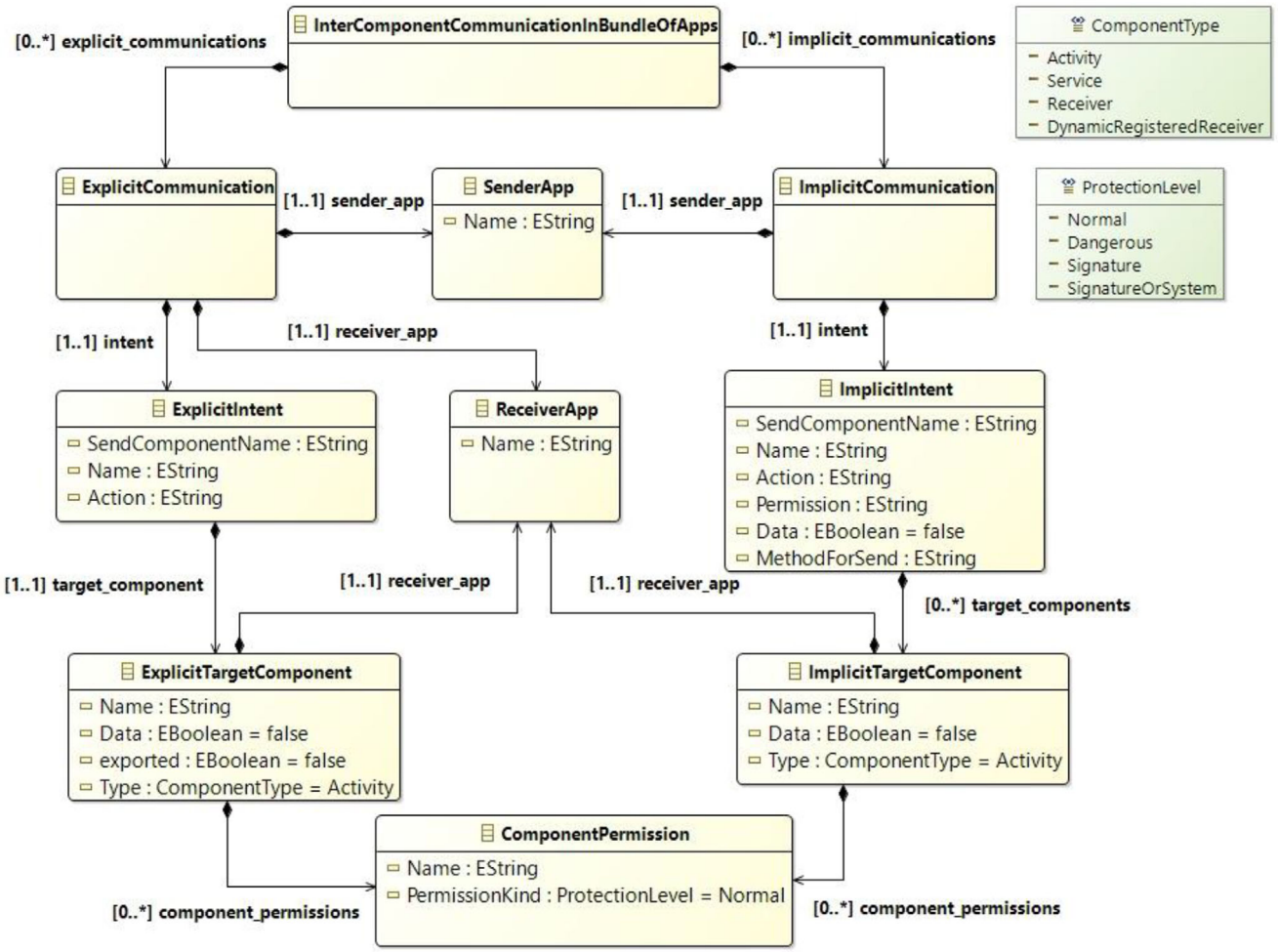

FIGURE 6 Intercomponent communication metamodel

Action, the general action that must be performed by the component receiving the intent. Each *ExplicitIntent* also has one *ExplicitTargetComponent* that specifies the receiver component for the intent.

ImplicitIntent represents all implicit intent objects that exist in a bundle of Android apps. According to the Android Application Security Aspects metamodel (the part related to the intent element), *ImplicitIntent* contains five attributes: *Name*, *SendComponentName*, *Action*, *Permission*, and *MethodForSend*. This element also has the *Data* attribute that indicates the intent has at least one or no Data object. Each *ImplicitIntent* also has a set of *ImplicitTargetComponent* that specifies the receiver app components, which are all components in the Android system that can receive the intent.

5.3 | Approach overview

As depicted in Figure 7, the proposed approach works at two levels. First, at the *Application level*, it receives a bundle of Android apps, then, at the *Model level*, it generates useful high-level representations of these apps and their interactions. The main goal of this approach is to provide a better comprehension of the Android system through relevant model-based representations. These representations (models) can then be employed for a variety of purposes, including inter-app security analysis. Note that, At the *Application level*, both the APK package and source code of the corresponding application are allowed. For the case of the APK package, it will be decompiled to obtain its source code using the Jadx[44] tool.

As indicated at the *Model level*, the proposed approach consists of three main phases: (1) *Model Discovery* that extracts the initial models from each app without losing any information. These models are detailed enough to be considered as the starting point of various MDRE scenarios, including inter-app security analysis. (2) *Transformation and Integration*

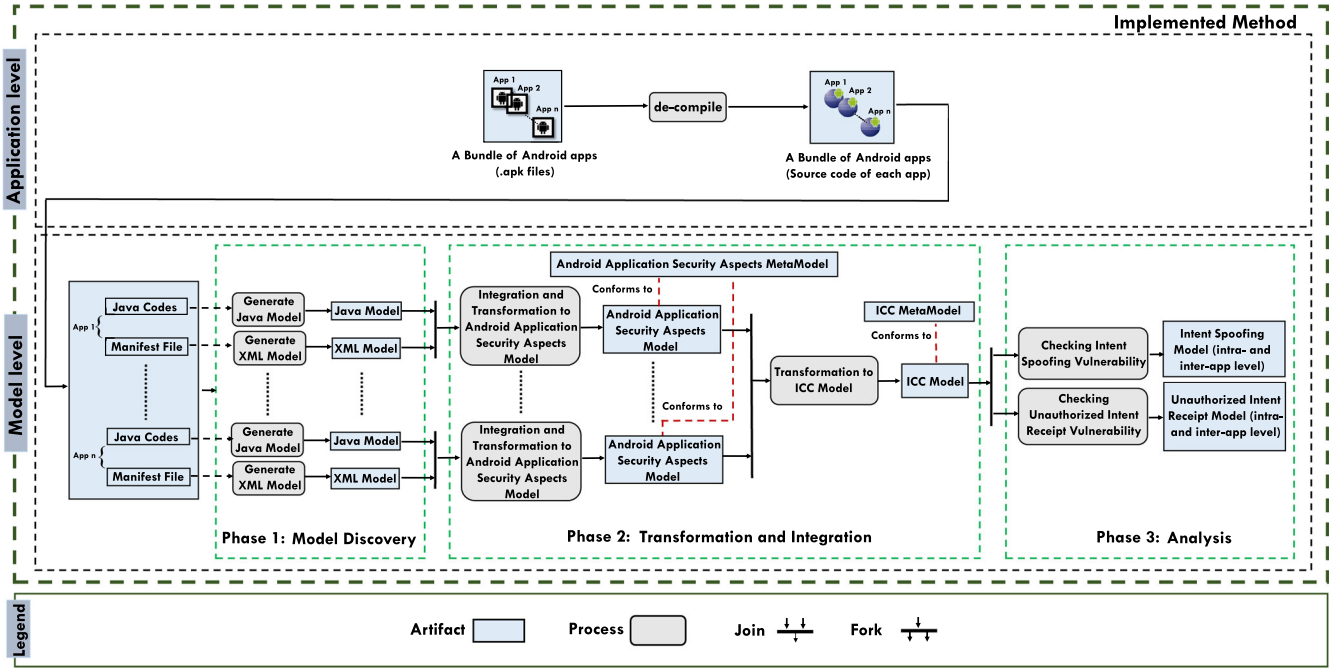

FIGURE 7 The proposed approach

that uses the chain of model manipulation techniques to transform initial models into more manageable representations. This transfer is done by deleting the details that are not relevant to inter-app security analysis. This process, which results in a higher level of abstraction, helps understand Android apps and their interactions better. (3) *Analysis* that uses the processed models in the previous phase to inter-app security analysis, and finally generates the results in the form of XMI models. In the following, each phase is described in more detail.

5.3.1 | Model discovery phase

Most static analysis methods have used frameworks and tools such as Soot, WALA [‡], and JPF [§] that perform their analysis based on an IR of the code. These tools have various limitations, including considering only a part of the specification of the Android platform.[12] Therefore, the purpose of this phase (i.e., phase 1 of Figure 7) is to provide a comprehensive IR of Android applications. The idea to reach this purpose is to transform from the low-level heterogeneous apps into homogeneous models. In this transition, the artifacts of apps, including Java code and Manifest files, are displayed as a set of interrelated models. For each app, the initial models (i.e., the XML model and the Java model) are obtained through model discoverers provided by the MoDisco tool[¶]. The created models are considered as inputs for the Transformation and Integration phase. These obtained models represent apps at the same level of abstraction to ensure that no information in this phase is lost.

5.3.2 | Transformation and integration phase

In this phase (i.e., phase 2 of Figure 7), the comprehension of the generated initial models (i.e., the created models in the previous phase) takes place by raising the abstraction level of these initial models using model-to-model (M2M) transformations written in ATL language[30] and obtaining higher-level representations of the Android system.

[‡]https://researcher.watson.ibm.com/researcher/view_page.php?id=7238
[§]https://github.com/javapathfinder/jpf-core/wiki
[¶]https://www.eclipse.org/MoDisco/

Algorithm 1. Extracting ICCs

Input: *AASAM-SET*: A set of Android Application Security Aspects Models
Output: *ICCM*: ICC Model

1: // **Explicit Communications Extractor - See Sec. 5.3.2**
2: *ExplicitCommunications* ← *extractAllExplicitIntents (AASAM-SET)*
3: *AppComponents* ← *extractAllComponents (AASAM-SET)* ∪ *extractAllDynamicRegisteredComponents (AASAM-SET)*
4: **for** each *EI* = < *SCN*, *TCN*, *N*> ∈ *ExplicitCommunications* **do**
5: *SenderApp* ← *extractApplication (SCN)*
6: **if** *TCN* ∈ *AppComponents* **then**
7: *ExplicitTargetComponent* ← *getAppComponent (AASAM-SET, TCN)*
8: *ReceiverApp*← *getApplication (AASAM-SET, TCN)*
9: **else**
10: *ExplicitTargetComponent* ← ∅ // There is no receiver component for this intent in the bundle
11: *ReceiverApp*← ∅ // There is no receiver app for this intent in the bundle
12: **end if**
13: **end for**
14: // **Implicit Communications Extractor - See Sec. 5.3.2**
15: *ImplicitCommunications* ← *extractAllImplicitIntents (AASAM-SET)*
16: **for** each *II* = <*SCN*, *N*, *A*, *C*, *D*> ∈ *ImplicitCommunications* **do**
17: *SenderApp* ← *extractApplication (SCN)*
18: // **Perform Intent Resolution - See Algorithm 2**
19: *ImplicitTargetComponents* ← *intentResolution (II)*. *getImplicitTargetComponents()*
20: **end for**

As depicted in Figure 7 (i.e., phase 2), first, for each app, the security information of initial models is extracted and gathered into a single security model called Android application security aspects using an M2M transformation. This generated model conforms to the proposed metamodel shown in Figure 5. The ATL code for this transformation includes 31 rules and 27 helpers. 16 rules and 8 helpers are considered to extract the information from the XML model, whereas the remaining 15 rules and 19 helpers are implemented to extract the security information from the Java model. Then, all Android application security aspects models are received as input models to be transformed into a single model called ICC. This model represents all potential intent-based communications at the intra- and inter-app levels. The ICC model conforms to the proposed metamodel shown in Figure 6. In the following, the process of modeling these domains (i.e., the *Transformation to ICC Model* process in Figure 7) is described.

5.3.3 | Extracting ICC model

The model-driven chain of this step includes an M2M transformation, written in ATL language, as shown in Figure 8. The ATL code for this transformation includes 17 rules and 28 helpers. In this transformation, a set of Android application security aspects models, the Android application security aspects metamodel, and the ICC metamodel are received as inputs, and the ICC model is obtained as output. The steps of this modeling are shown in Algorithm 1. As depicted in this algorithm, the Extracting ICC Model performs two major steps to extract the ICC model from a bundle of Android applications: *Explicit Communication Extractor* (lines 1–13) and *Implicit Communication Extractor* (lines 14–20), which are described in the following.

(1) Explicit communication extractor.
As shown in Algorithm 1, to model the explicit communication domain, first (line 2), all explicit intents are extracted from the desired Bundle of Android apps. According to the Android Application Security Aspects metamodel (the part related to the intent element and its references), an explicit intent is considered as a tuple *EI* = <*SCN*, *TCN*, *N*>, where *SCN* is the SendComponentName attribute, *TCN* is the TargetComponentName attribute, and *N* is the Name attribute.

Algorithm 2. Intent resolution process

Input: $II = <SCN, N, A, C, D>$: An implicit intent in the bundle of Android app
Output: *ImplicitTargetComponents*: A set of target components in the Android system (the bundle of Android apps)

```
1:  ImplicitTargetComponents ← { }
2:  // ActionTest&CategoryTest&DataTest
3:  if A ≠ ∅ & C ≠ ∅ & D ≠ ∅ then
4:      ImplicitTargetComponents ← ActionTestCategoryTestDataTest (II)
5:  else
6:      // ActionTest&CategoryTest
7:      if A ≠ ∅ & C ≠ ∅ & D = ∅ then
8:          ImplicitTargetComponents ← ActionTestCategoryTest (II)s
9:      else
10:         // ActionTest&DataTest
11:         if A ≠ ∅ & C = ∅ & D = ∅ then
12:             ImplicitTargetComponents ← ActionTestDataTest (II)
13:         else
14:             // CategoryTest&DataTest
15:             if A = ∅ & C ≠ ∅ & D ≠ ∅ then
16:                 ImplicitTargetComponents ← CategoryTestDataTest (II)
17:             else
18:                 // ActionTest
19:                 if A ≠ ∅ & C = ∅ & D = ∅ then
20:                     ImplicitTargetComponents ← ActionTest (II)
21:                 else
22:                     // CategoryTest
23:                     if A = ∅ & C ≠ ∅ & D = ∅ then
24:                         ImplicitTargetComponents ← CategoryTest (II)
25:                     else
26:                         // DataTest
27:                         if A = ∅ & C = ∅ & D ≠ ∅ then
28:                             ImplicitTargetComponents ← DataTest (II)
29:                         else
30:                             ImplicitTargetComponents ← ∅
31:                         end if
32:                     end if
33:                 end if
34:             end if
35:         end if
36:     end if
37: end if
```

For each explicit intent, (line 5), the Android app whose component creates this intent is considered as the sender app (i.e., *SenderApp* in the ICC metamodel). Since an explicit intent must be delivered to the app component specified by the TargetComponentName attribute of the intent, (lines 6–12), if there exists a component in the Android system that is declared by the explicit intent, this component is considered as the receiver component (i.e., *ExplicitTargetComponent* in the ICC metamodel). The Android app whose component is declared by the explicit intent is considered as the receiver app (i.e., *ReceiverApp* in the ICC metamodel).

(2) Implicit communication Extractor.
As shown in Algorithm 1, to model the implicit communication domain, all the implicit intents in the Android system are extracted (line 15). According to the Android Application Security Aspects metamodel (the part related to the intent element and its references), an implicit intent is considered as a tuple $II = <SCN, N, A, C, D>$, where

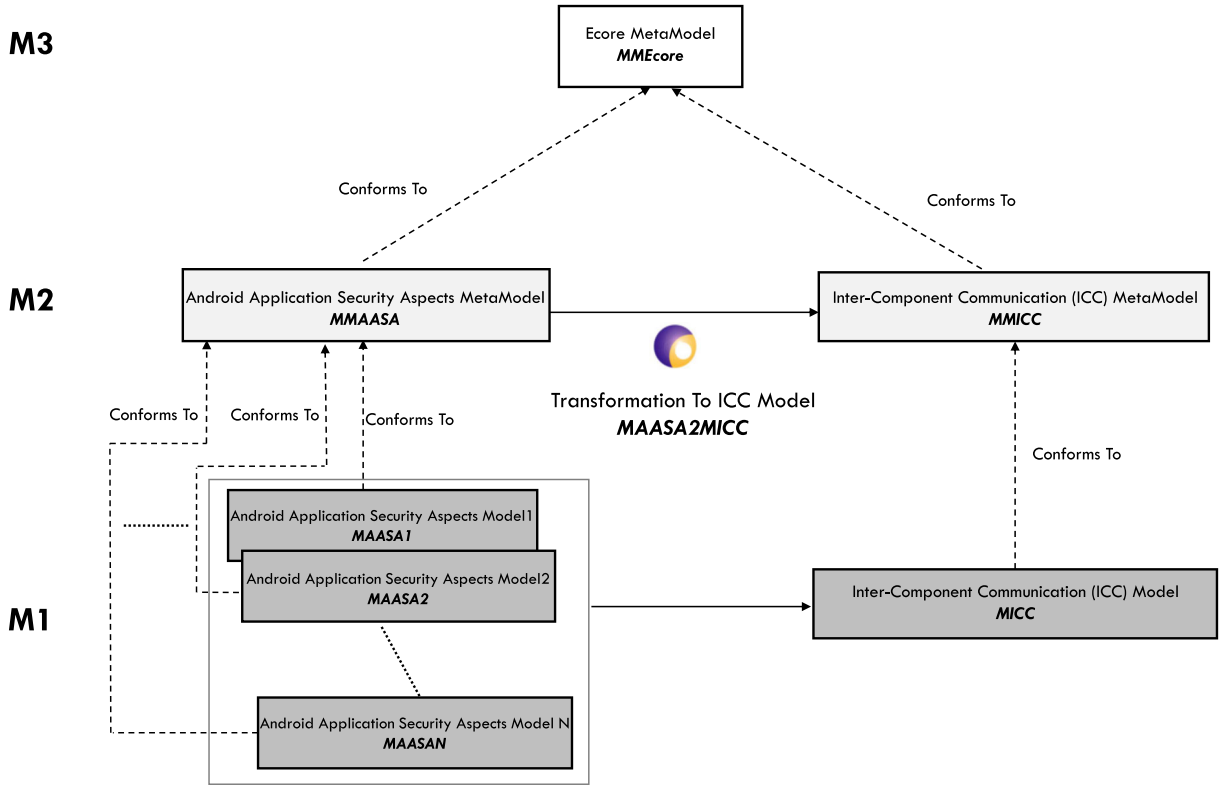

FIGURE 8 Transforming Android application security aspects models into an intercomponent communication model

SCN is the SendComponentName attribute, *N* is the Name attribute, *A* is the Action attribute, *C* is a set of categories, and *D* is a set of data. According to the specification of each intent, first, the Android app whose component creates this intent is considered as the sender app (i.e., *SenderApp* in the ICC metamodel) (line 17).
Then, in line 19, the Intent Resolution process[45] is used to extract and determine the receiver app(s) and the receiver component(s) (all components in the Android system that can receive and handle this intent). The concept of Intent Resolution and its implementation steps are explained in the following.

5.3.4 | The intent resolution process

When the Android system receives an implicit intent, for example, an implicit intent to launch an Activity, the Android system searches to find the best Activity or Activities for the desired intent. To determine the receiver component(s), the Android system compares the specification of the intent with the intent filters in all available Activity components.[45] This comparison is made through the following tests:

(1) Action test. The action specified in intent must match one of the actions in the intent filter of the desired component.
(2) Category test. The intent filter for the desired component must have the categories specified in the intent.
(3) Data (both URI and Data type) test. Depending on the structure of the data, the intent filter of the desired component must support the data in the intent.

The steps considered to implement this process using the ATL and OCL rules are shown in Algorithm 2. This function receives an implicit intent as input and returns app components that can receive and handle this intent.

As depicted in this Algorithm, depending on what features the intent has (i.e., action, category, and data), the Intent Resolution Process performs the relevant tests to identify all components of the Android system that can receive and handle this intent. Listing 6 shows an ATL helper, part of *Transformation to ICC Model* code (i.e., *MAASA2MICC* ATL code in Figure 8), that implements the Action test for an implicit intent and identifies a set of Activity components that can receive and handle this intent.

```
---test OnlyAction for Activity:
helper context MMAASA!Intent def:TestActionForActivity:Sequence(MMAASA!Activity)=
if MMAASA!Component.allInstances()->select(c | not c.oclIsTypeOf(MMAASA!ContentProvider))
   .notEmpty() then
  if MMAASA!Activity.allInstances()->select(c | not c.filters.oclIsUndefined()).notEmpty
   () then
  if not thisModule.stringOfIntentAction2FilterAction.get(self.Action).oclIsUndefined()
   then
     if not thisModule.IdentifyReceiverComponentActivity(thisModule.
   stringOfIntentAction2FilterAction.get(self.Action)).oclIsUndefined() then
          thisModule.IdentifyReceiverComponentActivity(thisModule.
   stringOfIntentAction2FilterAction.get(self.Action))->flatten()->collect(c | thisModule
   .createTargetComponentForActivity(c))
      else
        OclUndefined
    endif
   else
    if not thisModule.IdentifyReceiverComponentActivity(self.Action).oclIsUndefined()
  then
     thisModule.IdentifyReceiverComponentActivity(self.Action)->flatten()->collect(c |
  thisModule.createTargetComponentForActivity(c))
    else
     OclUndefined
    endif
   endif
  else
   OclUndefined
  endif
else
OclUndefined
endif;
```

Listing 4: ATL helper to perform Action test on Activity components for an implicit intent

Note that the two algorithms (Algorithm 1. Extracting ICCs and Algorithm 2. Intent resolution process) have been implemented precisely and comprehensively to make VAnDroid2 a generic framework for supporting multiple security issues, including inter-app security analysis. Due to the lack of space in this paper, the details of the ATL and OCL rules implemented for these algorithms are explained in a technical report.[46]

5.3.5 | Analysis phase

By gathering all potential communications between Android apps and representing them in a single model, it is possible to automatically and effectively perform security analysis at the intra- and inter-app levels. In this paper, we focus on two prominent inter-app vulnerabilities called Intent Spoofing and Unauthorized Intent Receipt.

Intent Spoofing is an ICC vulnerability. Definition 1 is used to identify all potential communications that can cause this type of security vulnerability. In this security threat, the malicious component (*c1* in Definition 1) can communicate with a public component (*c2* in Definition 1), while this communication is not expected. If the public component trusts the incoming (received) intent without performing the required security checks, the malicious app component can cause this app component to perform malicious actions. As it was previously discussed, according to the type of the victim component (c_2.*Type* in Definition 1), there are three types of Intent Spoofing:[23] (1) *Malicious Activity launch*, (2) *Malicious Service launch*, and (3) *Malicious Broadcast injection*.

Definition 1 (**Intent spoofing**). Let *BApps* be a set of benign apps, *MApps* be a set of malicious apps, *AppBundle* be a set of benign and malicious apps (i.e., *AppBundle* = *BApps* ∪ *MApps*), *C* be a set of all three kinds of components (Activity, Service, and Broadcast Receiver), *TargetComps* be a set of all public (exported) app components of *C* that can receive an intent, c_1 and c_2 be two members of *C* (i.e., $c_1, c_2 \in C$), *P* be a set of all four kinds of permissions (normal, dangerous, Signature, and SignatureOrSystem), *NormalPermissions* be a set of all permissions that their types are normal (i.e., *NormalPermissions* ⊆ *P*), c_2.*Permissions* be a set of permissions enforced by c_2, c_1 and c_2 can communicate together, and this communication can be done through an implicit intent or explicit intent. We say that communicate between c_1 and c_2 has inter-app Intent Spoofing vulnerability, if c_1 and c_2 do not belong to the same app and there is no permission for c_2 or all permissions in c_2. *Permissions* are normal permissions.

$$
\begin{aligned}
&IntentSpoofing_{attackType}(communication(c_1, c_2)) \equiv \\
&c_2 \in TargetComps \wedge \\
&c_1.app \in MApps \wedge c_2.app \in BApps \wedge \\
&c_1.app \neq c_2.app \wedge \\
&(c_2.Permissions = \emptyset \vee c_2.Permissions \subseteq NormalPermissions)
\end{aligned}
$$

$$
\text{where } attackType = \begin{cases} Malicious\ Activity\ launch, & \text{if } c_2.Type = Activity \\ Malicious\ Service\ launch, & \text{if } c_2.Type = Service \\ Malicious\ Broadcast\ injection, & \text{if } c_2.Type = Broadcast \end{cases}
$$

Unauthorized Intent Receipt is another ICC vulnerability. Definition 2 is used to identify all potential communications that can cause this type of security vulnerability. In this security threat, the malicious component (*c2* in Definition 2) can intercept an implicit intent (I_i in Definition 2) by introducing a filter that matches the sent intent. Therefore, the malicious app component can access all the data on the intent. As it was previously discussed, this ICC vulnerability can be of three types depending on the type of malicious app component ($c_2.Type$ in Definition 2):[23] (1) *Activity hijacking*, (2) *Service hijacking*, (3) *Broadcast theft*.

Definition 2 (**Unauthorized intent receipt**). Let *BApps* be a set of benign apps, *MApps* be a set of malicious apps, *AppBundle* be a set of benign and malicious apps (i.e., $AppBundle = BApps \cup MApps$), *C* be a set of all three kinds of components (Activity, Service, and Broadcast Receiver), *TargetComps* be a set of all public (exported) app components of *C* that can receive an implicit intent, c_1 and c_2 be two members of *C* (i.e., $c_1, c_2 \in C$), $communication_i(c_1, c_2)$ be an implicit communication between c_1 and c_2, I_i be an implicit intent that initiates the $communication_i(c_1, c_2)$, *P* be a set of all four kinds of permissions (normal, dangerous, Signature, and SignatureOrSystem), *NormalPermissions* be a set of all permissions that their types are normal (i.e., $NormalPermissions \subseteq P$), $I_i.Permission$ be the permission specified for I_i, We say that communication between c_1 and c_2 has inter-app Unauthorized Intent Receipt vulnerability, if c_1 and c_2 do not belong to the same app and there is no permission for I_i or $I_i.Permission$ be normal permission.

$$
\begin{aligned}
&UnauthorizedIntentReceipt_{attackType}(communication_i(c_1, c_2)) \equiv \\
&c_2 \in TargetComps \wedge \\
&c_1.app \in BApps \wedge c_2.app \in MApps \wedge \\
&c_1.app \neq c_2.app \wedge \\
&(I_i.Permission = \emptyset \vee I_i.P \in NormalPermissions)
\end{aligned}
$$

$$
\text{where } attackType = \begin{cases} Activity\ hijacking, & \text{if } c_2.Type = Activity \\ Service\ hijacking, & \text{if } c_2.Type = Service \\ Broadcast\ theft, & \text{if } c_2.Type = Broadcast \end{cases}
$$

An automated process is performed during this phase which includes three steps: measuring potential communications in the Android system, identifying communications that have the potential to cause a security threat, and finally presenting the results to the user using XMI models. The model-driven chain at this phase includes a set of M2M transformations written in ATL transformation language and OCL queries. Due to lack of space in this paper, the details of these transformations are described in a technical report.[46]

To better illustrate the phases of the proposed approach, consider the motivation example of Section 4. In the following, the results obtained in each phase are described. As discussed earlier, in the Model Discovery phase, for each app, the initial models (i.e., the XML model and the Java model) are obtained through MoDisco discoverers. These initial models generated for each app are shown in Figures 9 and 10. The Model Browser of MoDisco is used to display these models. As shown in Figures 9 and 10, in the left panel of the displayed models, the possible element types are specified (i.e., concepts related to the concerned metamodels), while in the right panel of the displayed models, the model elements themselves are shown. In the transformation and integration phase, first, for each app, the security information of initial models is extracted and gathered into a single security model called Android application security aspects. These models generated for each app are shown in Figure 11. As can be seen, these models represent the main specification and security structure of Android apps, including information about the app components. Then, by collecting these models, all potential ICCs

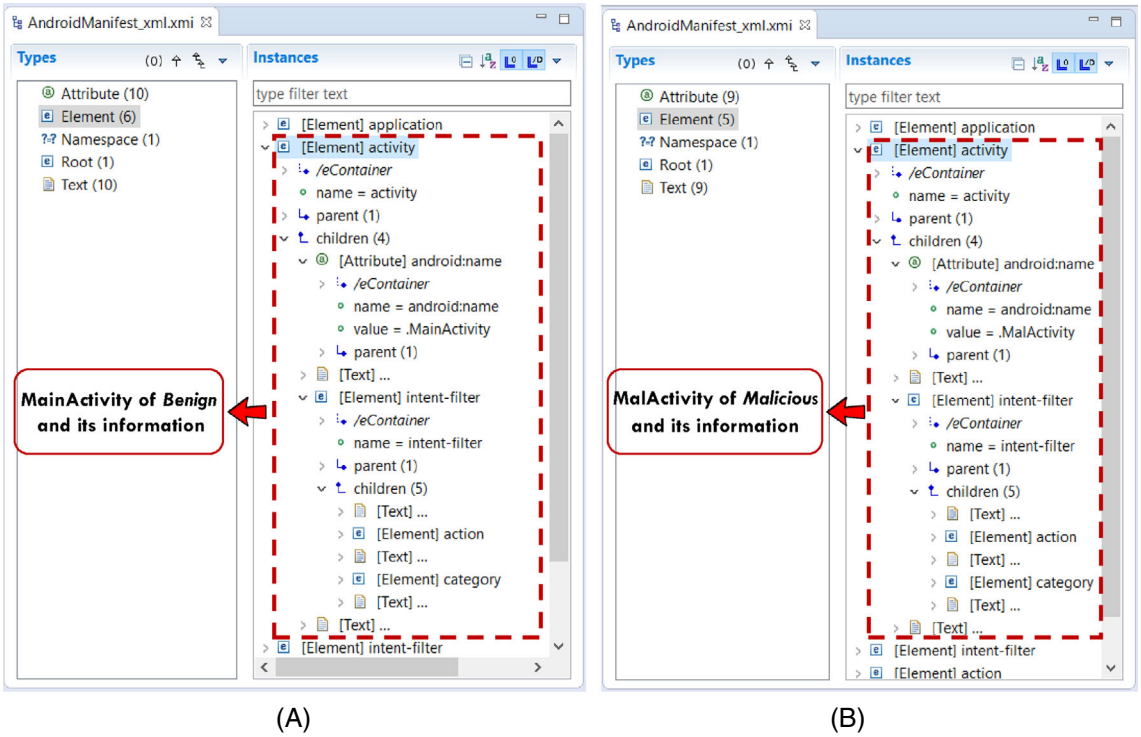

FIGURE 9 The XML models of the described apps in Section 4. (A) *Benign*; (B) *Malicious*

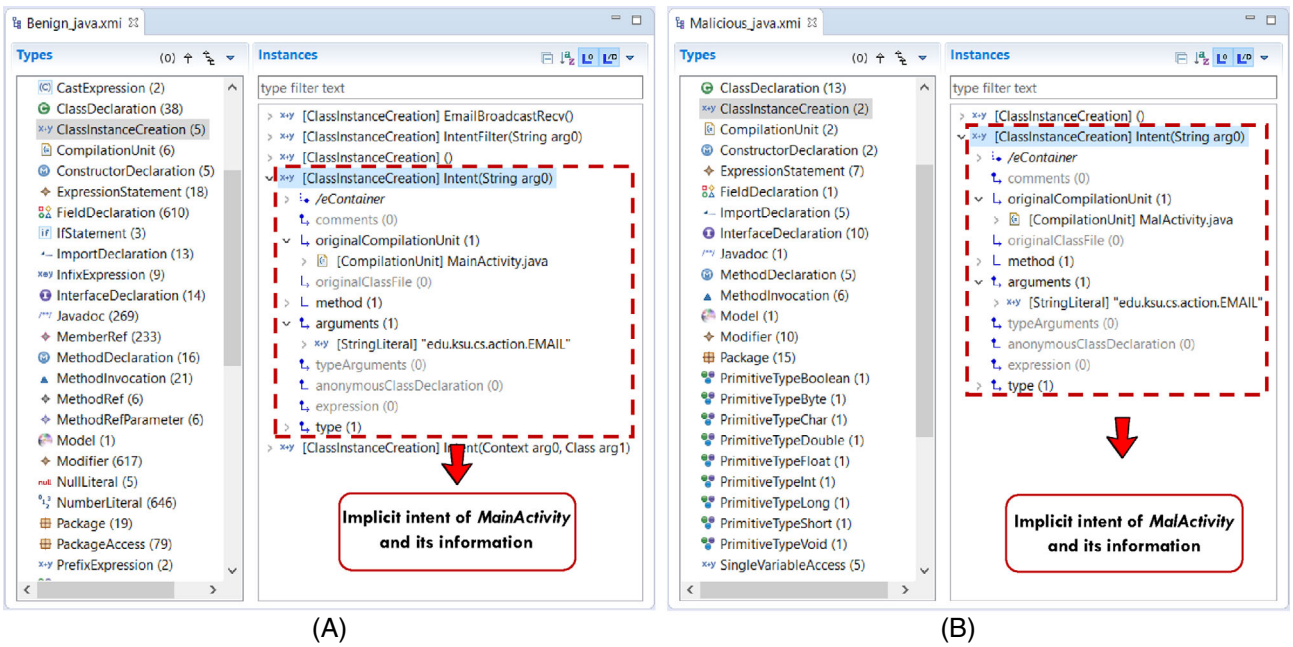

FIGURE 10 The Java models of the described apps in Section 4. (A) *Benign*; (B) *Malicious*

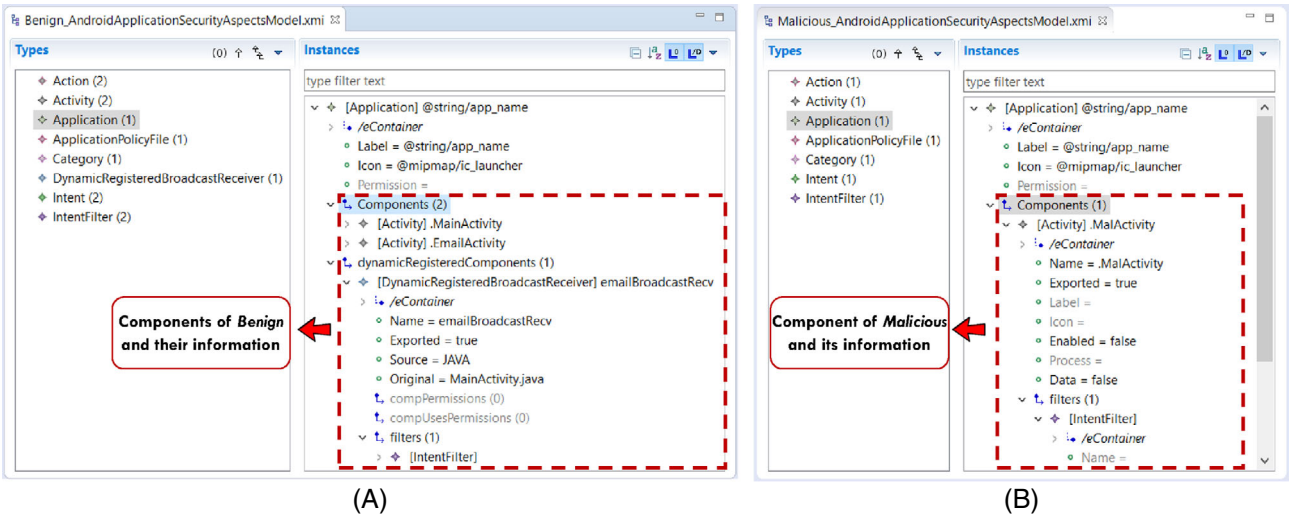

FIGURE 11 The Android application security aspects models of the described apps in Section 4. (A) *Benign*; (B) *Malicious*

(at intra- and inter-app levels) are extracted and integrated into a single model called ICC, which is shown in Figure 12. Since the intent sent by *Malicious* has the only action, the Action test is executed for it (line 18 in Algorithm 2). Therefore, all Broadcast Receivers are identified (shown in Figure 12) that can receive and handle this intent. Finally, in the Analysis phase, by performing formal analysis operations, the ICC vulnerability is identified. As depicted in Figure 13, the communication between *MalActivity* in *Malicious* and *emailBroadcastRecv* in *Benign* has a kind of Intent Spoofing vulnerability called Broadcast Injection.

5.4 | Incremental ICC analysis feature

In general, there are three types of ICC analysis approaches.[14] First, a pure program analysis approach, such as IccTA,[9] considers the entire system as a large program to perform the ICC analysis. Second, a hybrid yet nonincremental approach, such as COVERT,[10] divides the ICC analysis into two tasks: identifying security specification from each app and then examining the vulnerabilities of these specifications. Since these two approaches lack an appropriate way to consider the Android system changes, the ICC analysis must be repeated from the beginning for each app. In unstable environments, such as Android that apps are constantly added, removed, or updated, these approaches are often considered unscalable and impractical. Therefore, a third approach, incremental ICC analysis, such as FLAIR,[14] is proposed to use the results of the previous system analysis to optimize subsequent analysis and automatically update ICC analysis to respond to the system changes. In this paper, to further improve the performance and scalability of the proposed approach, the incremental ICC feature is considered. In the next section, details of the implementation of this feature are described.

5.5 | Tool support

To demonstrate the ability of the proposed approach to detect inter-app vulnerabilities, it is developed as an Eclipse-based tool called VAnDroid2. Figure 14 shows the architecture of the tool implementation. Each phase of the proposed approach is implemented as a separate component so that each component provides the required input models for another component. The implementation of the metamodels (discussed in Sections 5.1 and 5.2) is based on the Eclipse Modeling Framework[#].

Any change to an Android system (i.e., deletion apps or addition apps) is considered in this tool to support the incremental ICC feature. To better illustrate this implementation, consider the app addition. When an Android app is added to

[#]https://www.eclipse.org/modeling/emf/

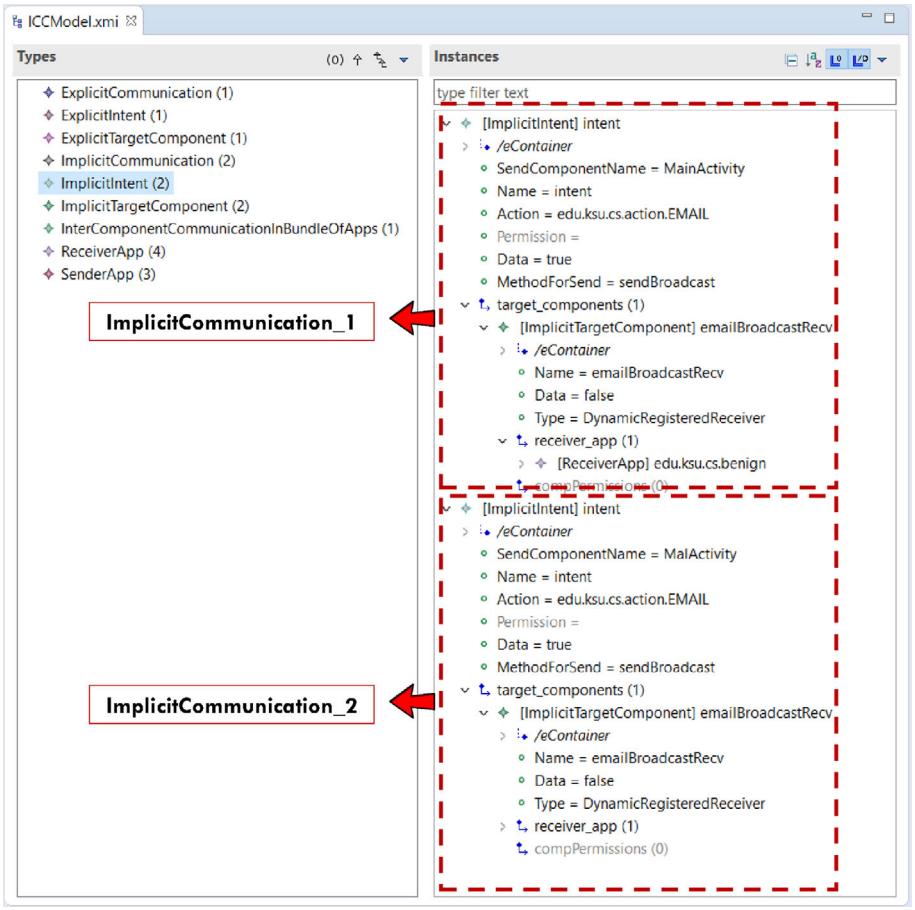

FIGURE 12 Intercomponent communication model of the described bundle in Section 4

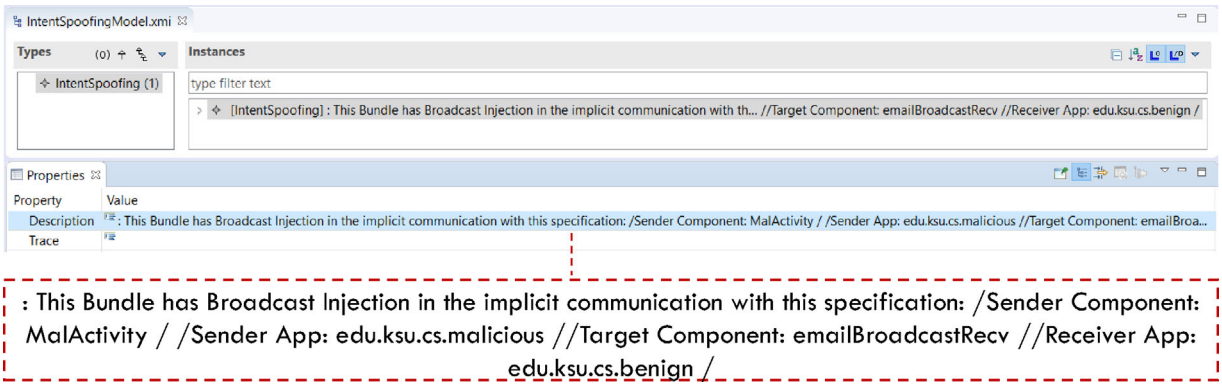

FIGURE 13 Evaluation model of the described bundle in Section 4

the system, according to Figure 14, *Initial Models Extractor* and *Android Application Security Aspects Models Extractor* are only executed for the new app, and all the analysis results of the previous system are used for these two components. *ICC Model Extractor* and *Inter-App Security Analyzer* are updated according to the specification of the new app for analyzing the new system.

6 | EMPIRICAL EVALUATION

To evaluate the ability of VAnDroid2 to detect inter-app vulnerabilities in real-world apps, VAnDroid2 has been applied to 10 bundles of real-world Android apps. To create these bundles, first, we constructed a dataset of benign, malicious, and

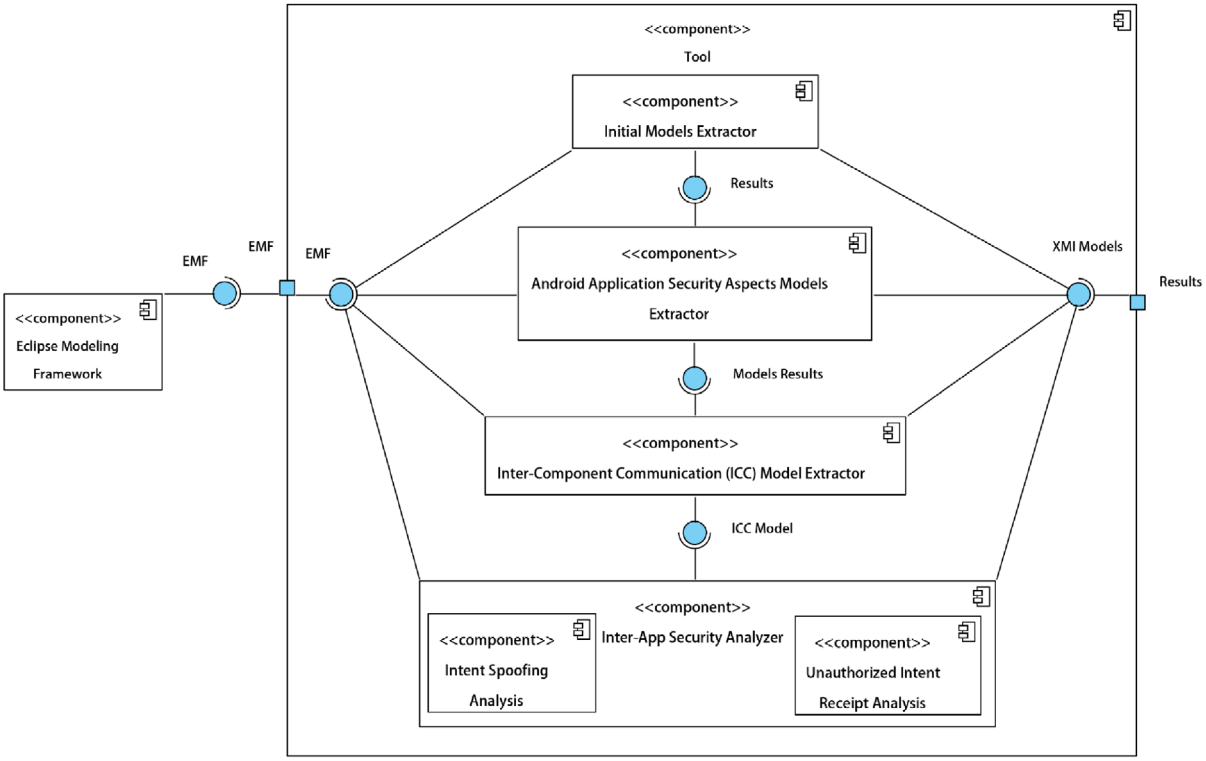

FIGURE 14 Component diagram of the implementation

vulnerable Android apps. Then, 10 app bundles, each containing 35 apps, have been randomly selected from our dataset. The following research questions are addressed in this evaluation:

- **RQ1 (Correctness)**: How reliable are the analysis results of VAnDroid2?
- **RQ2 (Scalability)**: Is VAnDroid2 capable of performing the inter-app analysis to identify vulnerabilities in real-world Android apps?
- **RQ3 (Run-time performance)**: Is VAnDroid2 practical in terms of execution performance?

Our experiments ran on a 3.7 GHz, Core i7 computer with 32 GB RAM.

In the following, a description of the provided dataset of real-world Android applications is given in Section 6.1. Then, the correctness, scalability, and run-time performance criteria are evaluated in Sections 6.2, 6.3, and 6.4, respectively.

6.1 | The dataset of real-world Android apps

We constructed a dataset of benign, malicious, and vulnerable Android applications, as shown in Table 2. For benign and malicious apps, AndroZoo[47] is considered. AndroZoo is a growing collection of millions of Android applications collected from various marketplaces such as Google Play ‖ and F-Droid **. From this collection, 500 benign apps (the creation date after December 2019), as well as 300 malicious apps (the creation date after December 2018), are selected. In addition, 257 vulnerable apps are considered from four Android application vulnerability benchmarks, including DroidBench,[48] ICC-Bench,[49] Ghera,[1] and UBCBench.[50]

According to the mobile app statistics in recent two years,[51,52] on average, 40 apps in 2020 and 30 apps in 2021 are installed on the user's devices per month. Therefore, in this evaluation, 10 bundles of apps, each containing 35 apps, are created randomly from the provided dataset. Each bundle contains 29 benign apps, 3 malicious apps, and 3 vulnerable

‖https://play.google.com/store/apps/
**https://f-droid.org/

TABLE 2 Distribution of selected apps from various repositories that were considered in our dataset

Subject app	Numbers of apps	Repository
Benign	500	AndroZoo
Malicious	300	AndroZoo
Vulnerable	257	DroidBench
		ICC-Bench
		Ghera
		UBCBench
Total	**1057 unique Android apps**	

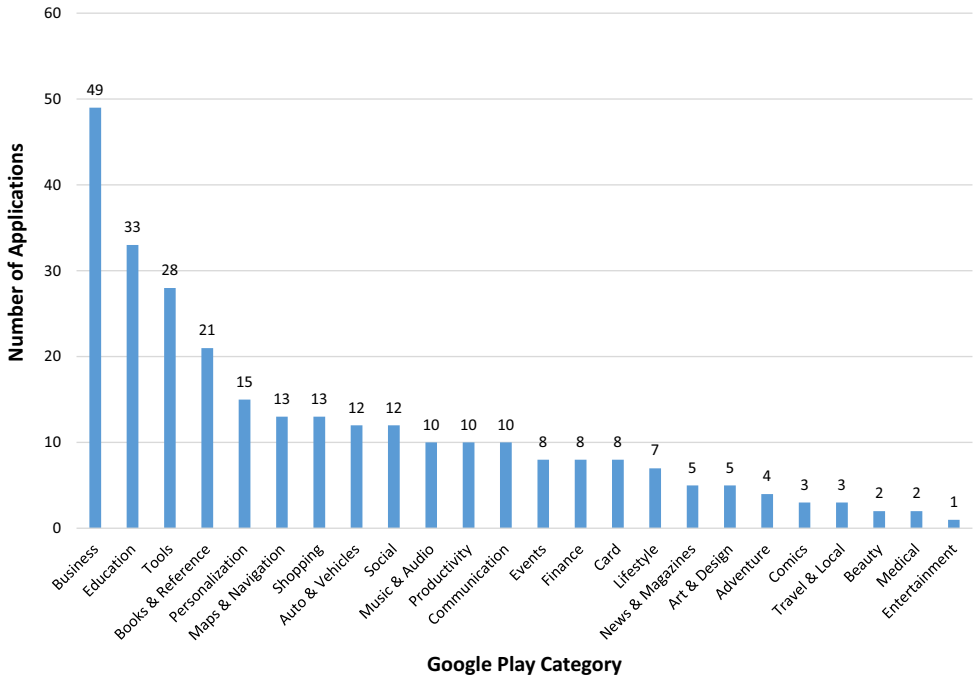

FIGURE 15 Histogram of categories of apps selected from Google Play (282 Android apps)

apps. Figure 15 illustrates the distribution of benign apps that were used in the app bundles and belong to the Google Play repository. Note that out of 290 benign apps used in the app bundles, 282 apps belong to the Google Play repository. As can be seen in Figure 15, these applications are sufficiently diverse in terms of categories. Therefore, the selected benign apps from Google play are varied across application domains. Also, according to the statistics of the most popular Google Play app categories in 2022,[53] most of the popular categories exist in our evaluation. As depicted in Figure 16, these apps vary in terms of 5-star ranking. Therefore, these app bundles simulate the apps installed on the Android devices of the user, and we consider them to perform 10 independent experiments.

6.2 | RQ1 (Correctness)

According to the definition given by Pressman and Maxim,[54] correctness indicates the extent to which a software product satisfies its objectives. This criterion determines the correctness of the approach used in the VAnDroid2 tool and to what extent the analysis results of this tool (i.e., inter-app vulnerability reports) can be reliable.

To examine the correctness criterion, it is necessary to consider an Android app vulnerability benchmark that proposed known vulnerabilities in its bundles. For this purpose, the Ghera repository[1] is considered. To specify the reason for the selection of the Ghera repository, the main characteristics of this repository are described in the following.

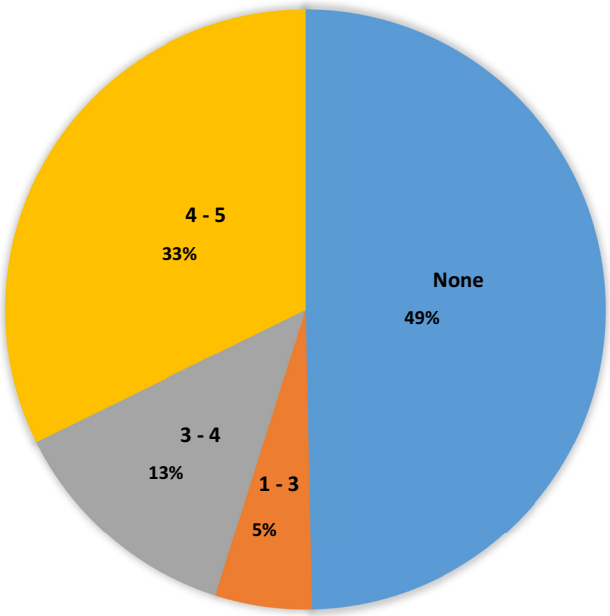

FIGURE 16 Ratings of selected apps from Google Play according to their 5-star ranking (282 Android apps)

Ghera[1] is a growing repository of Android apps (benchmarks) that captures known vulnerabilities in Android applications in various categories, including *Crypto*, *Networking*, *NonAPI*, *Permission*, *Storage*, *System*, *Web APIs*, and *ICC*. This repository contains several bundles of Android apps. In each bundle, there is a benign app that has a specific vulnerability and a malicious app that exploits benign to establish a malicious communication. Since the vulnerabilities in the Ghera repository have already been reported in the literature and documented in Android documentation, these vulnerabilities are valid. These vulnerabilities can be examined by executing benign and malicious apps on Android emulators and devices. Therefore, they are exploitable and general. These vulnerabilities can occur on the Android app running on Android 5.1.1 to 8.1.[1]

Due to the characteristics of Ghera, to examine the correctness criterion, the Ghera repository is used. From this repository, only bundles related to ICC that contain Intent Spoofing or Unauthorized Intent Receipt are considered. The bundles are as follows:

(1) DynamicRegBroadcastReceiver-UnrestrictedAccess
(2) EmptyPendingIntent-PrivEscalation
(3) HighPriority-ActivityHijack
(4) ImplicitPendingIntent-IntentHijack
(5) StickyBroadcast-DataInjection
(6) UnprotectedBroadcastRecv-PrivEscalation

According to an empirical evaluation by Ranganath and Mitra,[1] COVERT[10] and DIALDroid[55] that claim to detect inter-app vulnerabilities failed to identify vulnerabilities in Ghera. Based on the results of applying VAnDroid2 to these bundles, this tool detected all the vulnerabilities in these bundles without any further reported security issues. Therefore, VAnDroid2 can correctly detect inter-app ICC vulnerabilities in Android applications, hence the approach conducted in this tool is reliable.

6.3 | RQ2 (Scalability)

Scalability indicates the ability of a software program to process an increasing number of elements as inputs.[56] The software system must be performed well while increasing the number of elements as inputs to examine this criterion.[57] According to this definition, in this evaluation, the ability of VAnDroid2 to deal with the following issues is examined:

TABLE 3 Summary of the results obtained from running VAnDroid2 over the selected app bundles

Bundle	Number of component	Number of intent		Number of intent filter	Number of communication domain		Number of component permission	Number of warning	
		Explicit	Implicit		Explicit	Implicit		Intent spoofing	Unauthorized intent receipt
Bundle 1	97	109	169	69	44	65	3	48	48
Bundle 2	97	82	121	64	43	101	5	31	93
Bundle 3	139	390	442	69	230	233	5	157	207
Bundle 4	90	63	140	62	25	38	—	21	20
Bundle 5	122	121	184	57	81	150	3	53	141
Bundle 6	163	191	235	61	134	43	1	28	28
Bundle 7	151	161	173	63	107	47	2	32	30
Bundle 8	188	154	182	89	80	130	13	79	120
Bundle 9	154	194	218	66	120	147	2	71	129
Bundle 10	137	159	267	70	86	196	4	123	173

- Android applications in various contexts
- Android applications with different components and resources
- Large applications (in terms of code size)
- Large number of applications

As explained before, we evaluated VAnDroid2 on 10 app bundles containing real-world Android applications to determine the ability of VAnDroid2 in inter-app ICC vulnerability detection. Table 3 shows the results of running VAnDroid2 on each bundle. The values in this table are the results obtained by applying VAnDroid2 on Android app bundles.

The *Component* column shows the total number of components (i.e., Activities, Services, Receivers, and Content Provides). The *Intent* column specifies the number of intents (both explicit and implicit). The *Intent Filter* column indicates the number of interfaces provided (i.e., intent filters). The *Communication Domain* column specifies all potential explicit and implicit interactions made by explicit and implicit intents, respectively.

The *Component Permission* column specifies the number of permissions enforced by components. The *Warning* column shows the number of warnings generated by VAnDroid2. Each of these warnings represents a unique interaction with all specifications (i.e., the sender and receiver components) that leads to an Intent Spoofing or Unauthorized Intent Receipt vulnerability. Note that the Intent Spoofing column is related to both explicit and implicit communication domains, but the Unauthorized Intent Receipt column is related to the implicit communication domain.

Consider bundle 10 in the last row, this bundle includes 137 app components, 159 explicit intents, 267 implicit intents, 70 intent filters, and 4 component permissions. As depicted in the *Communication Domain* column, out of 159 explicit intents, 86 explicit intents can make explicit inter-component communications in this app bundle. Also, out of 267 implicit intents, 196 implicit intents can make implicit ICCs in this app bundle. According to the *Warning* column, out of 282 ICCs, 123 communications can cause Intent Spoofing vulnerability. Out of 196 implicit ICCs, 173 communications can cause Unauthorized Intent Receipt vulnerability.

Table 4 shows the benign Android applications in each bundle according to the lines of code, the average number of Activities, the average number of intents, and the average evaluation time (in milliseconds). Note that the evaluation time is related to the time required for analyzing each app to extract the Android application security aspects model.

In Figure 17, we examined the execution time changes for the benign Android apps according to the three app-specific factors (i.e., the number of intents, the number of Activities, and LOC). As can be seen, when each of these factors increases significantly, the execution time does not increase significantly and is still an acceptable amount of time. The median evaluation time is 191.5 ms with the interquartile range of 832 ms, which indicates that VAnDroid2 can quickly analyze real-world Android applications.

TABLE 4 Benign Android apps in each bundle used in the evaluation

Line of code	Numbers of Apps	Average number of activities	Average number of intents	Average of evaluation time (ms)
0–10	5	2	0	111.6
10–1000	120	2	1	184.1
1000–10,000	49	4	10	320.6
10,000–100,000	112	6	26	1108.2
100,000–200,000	4	6	12	32648.5
0–200,000	**290**	**4**	**10**	**1010.63**

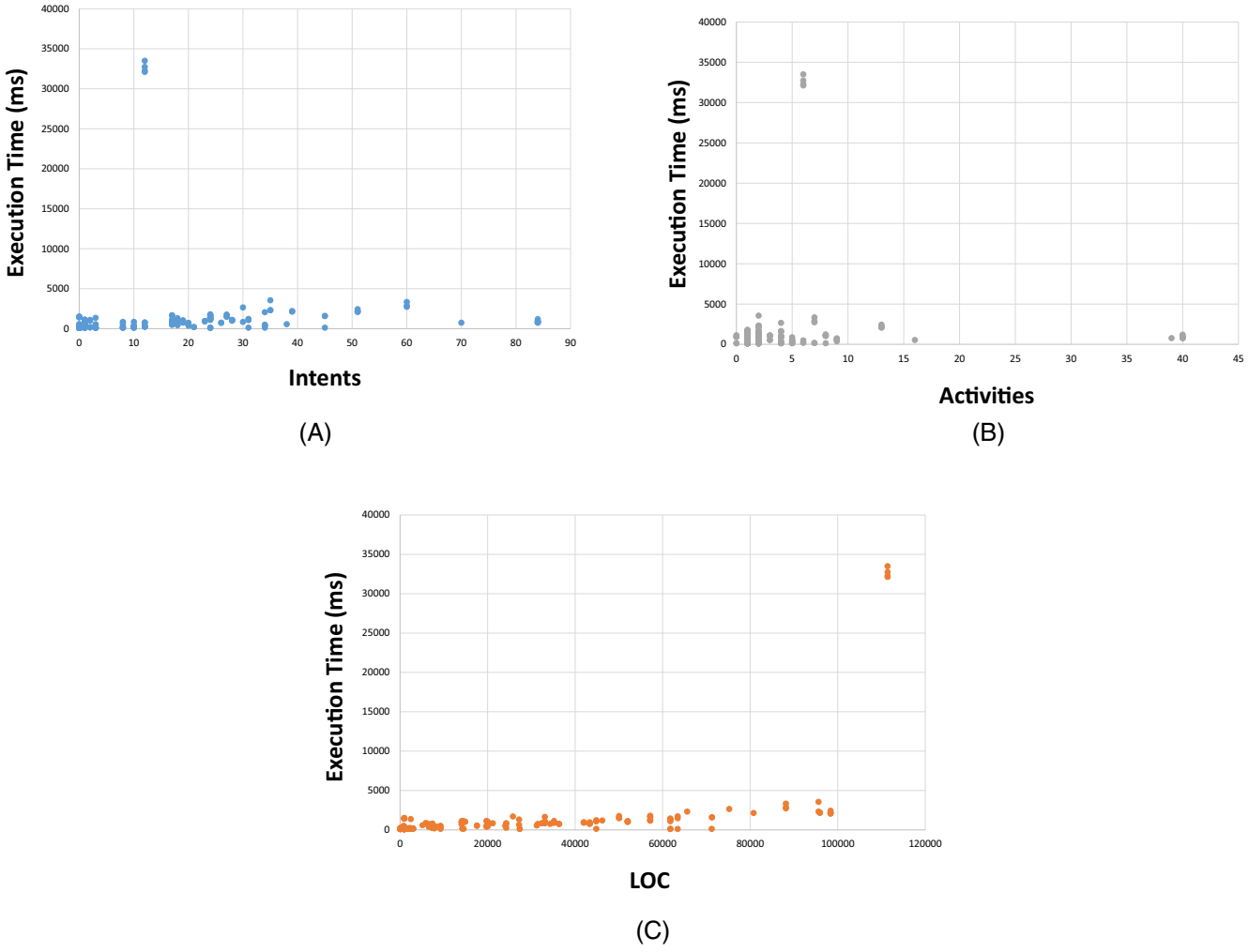

FIGURE 17 Scatterplot representing the execution time for the benign Android apps in each bundle according to (A) the number of intents, (B) the number of Activities, and (C) lines of code (LOC)

TABLE 5 Execution time of applying VAnDroid2 to selected bundles

	Execution time (ms)		
Bundle	**Phase 1**	**Phases 2 and 3**	**Total time**
Bundle 1	18,655	9765	28,420
Bundle 2	11,549	10,391	21,940
Bundle 3	17,440	9663	27,103
Bundle 4	48,916	9515	58,431
Bundle 5	15,842	9610	25,457
Bundle 6	13,867	8979	22,846
Bundle 7	15,578	9446	25,024
Bundle 8	50,363	10,733	61,092
Bundle 9	50,993	10,802	61,795
Bundle 10	52,128	10,690	62,818

The results show that VAnDroid2 is able to automatically detect inter-app vulnerabilities in bundles of real-world apps in various contexts (depicted in Figure 15), with different components and resources (depicted in Tables 3 and 4), with large apps (i.e., in terms of code size as depicted in Table 4), and with new versions of Android API (according to the creation date of apps which are after December 2019). Therefore, VAnDroid2 has been able to support the scalability criterion in the detection of ICC vulnerabilities of Android apps.

6.4 | RQ3 (run-time performance)

As described in Section 5.4, to further improve the performance and scalability of the proposed approach, incremental ICC analysis is also considered. In this section, first, the execution time of applying VAnDroid2 to the 10 bundles is examined. Then, the execution time of VAnDroid2 according to the incremental ICC analysis feature is evaluated.

6.4.1 | Results for selected bundles

Since the first phase of VAnDroid2 is based on MoDisco, As depicted in Table 5, we measured the execution time of phase 1 of Figure 7 and the execution time of phases 2 and 3 of Figure 7, separately. As can be seen, VAnDroid2, in less than a few minutes, is able to analyze and identify inter-app ICC vulnerabilities in bundles of real-world apps that have different components and interactions.

6.4.2 | Results for incremental ICC analysis

Figure 18 shows the execution time of VAnDroid2 with a gradual increase in the number of apps for selected bundles. Note that, at first, we consider five apps in each bundle. Then, in each experiment, we add five new apps. Consider Bundle10 in Figure 18B; when its size reaches 15 apps, the execution time is reduced, and this indicates that the results of the analysis in the previous system were used in the revised system. Therefore, according to these two diagrams, VAnDroid2 has been able to support ICC incremental analysis to further improve the performance and scalability of the proposed approach.

7 | DISCUSSION

According to the evaluation results, VAnDroid2, as a framework for detecting security issues in IAC, can be used by developers, security analysts, and researchers of Android applications. In the following, to discuss the strengths and weaknesses of VAnDroid2, in Section 7.1, a breakdown of the ICC vulnerabilities reported by VAnDroid2 is compared with the work of Chin et al.[23] In Section 7.2, VAnDroid2 is compared with the related work that are proposed in the static

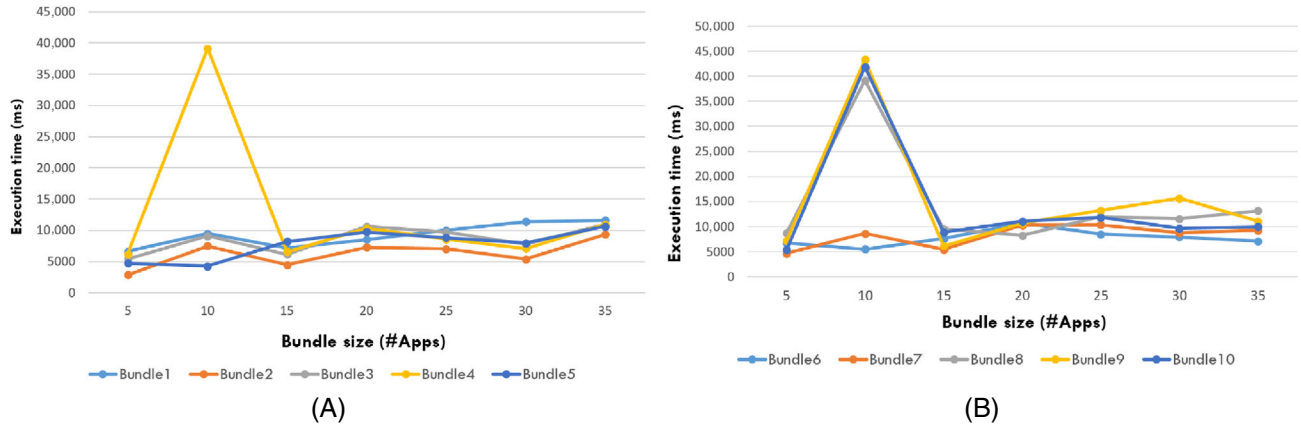

FIGURE 18 Execution time versus the increasing size of analyzed bundles. (A) the first five bundles; (B) the last five bundles

TABLE 6 Obtained breakdown by Chin et al.[23] and by VAnDroid2

Kind of vulnerability	Chin et al.	VAnDroid2
Activity hijacking	57%	35%
Broadcast injection	14%	25%
Broadcast theft	12%	25%
Activity launch	12%	15%
Service hijacking	3%	0%
Service launch	1%	0%

Android analysis field (described in Section 3). In Section 7.3, VAnDroid2 is compared with several existing state-of-the-art tools related to ICC analysis of Android applications. The results of these comparisons indicate that the VAnDroid2 tool has been able to extract high-level representations (models) from the security specification and structure of Android apps without losing information. Also, this tool achieved more precision, recall, and F-measure than the other analysis tools. In Section 7.4, the (re)use and extension potential of VAnDroid2 for other types of ICC vulnerabilities are explained. Section 7.5 provides a tabular comparison between VAnDroid2 and the original VAnDroid framewor[19] to specify in detail the significance of the extensions made in VAnDroid2. Finally, in Section 7.6, limitations and threats to the validity of VAnDroid2 are discussed.

7.1 | Breakdown of the discovered vulnerabilities

Chin et al.[23] discussed several ICC attacks in Android inter-app communication and proposed a breakdown of these attacks in real-world Android apps. According to the reported breakdown by Chin et al.,[23] in the second column of Table 6, the Activity hijacking vulnerability is the most common inter-app vulnerability, because Activity uses the message passing mechanism through intent more than other components. According to the discovered vulnerabilities by VAnDroid2 (i.e., the *Warning* column in Table 3), the obtained breakdown, in the third column of Table 6, is similar to the breakdown obtained by Chin et al.,[23] and Activity hijacking is the common vulnerability in Android applications.

7.2 | Comparing with related work

Table 7 presents a comparison between the proposed approach and the related studies proposed in the static Android analysis field (introduced in Section 3). This comparison is based on the granularity of security threats, level of security threats, detected vulnerabilities, conducted technique, and the framework used for code representation. As can be seen,

TABLE 7 A comparison between VAnDroid2 and related work

		ComDroid[23]	CHEX[35]	Epicc[7]	FlowDroid[8]	DidFail[39]	COVERT[10]	IccTA[9]	DroidSafe[37]	ICCMATT[40]	Amandroid[11]	IIFA[38]	VAnDroid2
Granularity of security threats	Intracomponent	●	●	●	●	●	●	●	●	●	●	●	●
	Intercomponent	●	●	●	○	●	●	●	●	●	●	●	●
Level of security threats	Intra-app	●	○	●	○	●	●	●	●	●	●	●	●
	Inter-app	○	○	○	○	●	●	◑	○	○	●	●	●
Detected vulnerability	Sensitive information leakage	○	●	○	●	●	◑	●	●	●	●	●	◑
	Intent spoofing	●	○	●	○	○	●	○	○	○	○	○	●
	Unauthorized intent receipt	●	○	●	○	○	●	○	○	○	○	○	●
	Permission misuse	○	○	○	○	○	●	○	○	○	○	○	○
	Data validation	○	○	○	○	○	○	○	○	○	●	○	○
Conducted technique	Taint Analysis	○	○	○	●	●	◑	●	●	●	●	●	◑
	Code Instrumentation	○	○	○	○	●	○	●	●	○	○	○	○
	Type/Model Checking	○	○	○	○	○	●	○	○	●	○	●	●
	Data Flow Analysis	○	○	●	●	○	◑	●	●	○	●	●	◑
	Reachability Analysis	●	●	○	○	○	○	○	○	○	○	○	○
	Model Driven	○	○	○	○	○	○	○	○	○	○	○	●
Code representation	Jimple	○	○	●	●	●	●	●	●	○	○	○	○
	Smali	○	●	○	○	○	○	○	○	○	○	●	○
	Other	●	○	○	○	○	○	○	○	●	●	○	●

Abbreviations: ●, Full support; ○, No support; ◑ Partial support.

most of the solutions have been done to address the detection of sensitive information leakage in Android applications, and some analysis approaches have been proposed to identify ICC vulnerabilities, such as Intent Spoofing and Unauthorized Intent Receipt. The majority of related studies are considered a single Android application. These studies do not support the ICC analysis at the inter-app level. As depicted in this table, the VAnDroid2 tool identifies two prominent ICC vulnerabilities (i.e., Intent Spoofing and Unauthorized Intent Receipt) at both intra-app and inter-app levels.

The *Conducted Technique* row of Table 7 delineates the analysis techniques that are conducted by VAnDroid2. The taint analysis technique tracks the flow of sensitive data within the programs. As previously explained, Android allows ICC at both intra-app and inter-app levels mainly through intent messages. These intent objects may transfer tainted data from one component or app to another.[16,55] Since VAnDroid2 analyzes ICC at both inter-app and intra-app levels by focusing on the message passing mechanism through intent, it can detect sensitive ICCs that leak sensitive information. In this analysis, a leak is considered as a communication between two components or two apps that originates in one class and ends in another class. Therefore, the taint analysis technique is partially supported by VAnDroid2.

VAnDroid2 conducts the ICC data flow analysis technique. In this analysis, a sensitive ICC is a communication from an ICC exit point such as bindService and startActivity to an ICC entry point such as getIntent and onActivityResult. This communication transfers an intent object that can be contained the sensitive data. In this sensitive ICC, the ICC exit leak identifies the sender app and the ICC entry leak identifies the receiver app.

Type and model checking are two methods for program verification. The purpose of type checking is to ensure that a program is safe in terms of type error. Model checking is a process that aims to check the status of the program to ensure that certain specifications are met.[6] VAnDroid2 first extracts domain-specific models of the security specifications of Android apps and their interactions. Then, in the analysis phase, the resulting models are analyzed against ICC vulnerabilities through a formal analysis process. Therefore, VAnDroid2 conducts the type/model checking technique in the ICC analysis.

In comparison with related studies, VAnDroid2 uses the MDRE technique to enable inter-app security analysis. In this tool, the structural complexity of the Android applications is reduced by extracting the security information of each Android app in the form of a single model. Due to the nature of these platform-independent models, the proposed approach does not depend on a specific API and considers features beyond the API in modeling. As a result, it will be able to support new versions of Android. Since one of the main principles of MDRE is to quickly create initial models of software artifacts without losing information,[17] in the proposed approach, these raw and completely accurate initial models are considered as IRs of the Android application code. These initial models are then considered as the input (starting point) for all the reverse engineering activities.

To summarize, since our study is based on MDRE, the main benefits of VAnDroid2 are as follows.

- Extensibility. Due to the use of metamodels, the ability to customize model-based components, and the clear decoupling of the result models from different phases, new features can be plugged into the developed tool.
- Full coverage. Full coverage of app artifacts can be provided through complementary representations of the app at different abstraction levels and considering different perspectives (i.e., metamodels).
- (Re)use and integration. Due to the clear separation of concerns in VAnDroid2, Android apps and their specifications displayed in the form of models are strictly separated, which further facilitates the reusability of such models.

7.3 | Comparing with existing tools

In comparison with other tools, the ability of VAnDroid2 to satisfy its goals and to generate reliable analysis results is examined. As explained earlier, one of the main goals of VAnDroid2 is to provide a comprehensive IR of the Android app to create high-level models without losing information. These models can also be used for an effective ICC and IAC security analysis. Therefore, first, in Section 7.3.1, VAnDroid2 is compared with IC3,[20] as the state-of-the-art Android static program analysis tool, in extracting specifications from the Android app. Then, in Section 7.3.2, VAnDroid2 is compared with three existing state-of-the-art Android static analyzers for ICC vulnerability detection.

7.3.1 | Comparing with IC3

To compare with IC3,[20] two benchmarks, DroidBench[48] and ICC-Bench,[49] are used. Table 8 shows the results of comparing the effectiveness of VAnDroid2 with IC3 in extracting security specifications from Android apps. As shown in this table, we counted the number of ICC values, including components, intent filters, and intents inferred by IC3 and VAnDroid2. Also, we measured the average execution time for each tool. The values that indicate the superiority of VAnDroid2 to IC3 are distinguished by the bolded text in this table.

According to Table 8, for apps of DroidBench, we observe that VAnDroid2 outperforms IC3 in inferring components, intent filters, and intents. The reason for these differences is related to the ability of tools in detecting all types of app components, including Activity alias and Broadcast Receivers. The *Echoer* app, related to IAC test cases of DroidBench, has

TABLE 8 Comparison of VAnDroid2 and IC3[20]

Tool	Number of components	Number of intent filters	Number of intent		Average of execution time (ms)
			Explicit	Implicit	
	DroidBench 2.0				
IC3	148	138	10	10	3.0
VAnDroid2	**172**	**160**	**12**	**56**	**1.7**
	ICC-Bench				
IC3	51	38	13	13	5.1
VAnDroid2	51	38	13	13	**1.3**

an Activity alias that IC3 could not detect. This component is considered as a separate component with its characteristics, including permissions and intent filters.[23] Therefore, this component should be considered in extracting information from the Android app. Furthermore, for some apps of DroidBench, IC3 could not detect Broadcast Receivers that were dynamically introduced in the Java code. This type of component is one of the most important app components and any app can send malicious intent to this component. It should be possible to identify all of these components in Android apps. As shown in Table 8, the average execution time of VAnDroid2 (1.7 ms) is considerably less than IC3 (3.0 ms). Therefore, for DroidBench, VAnDroid2 significantly outperforms the IC3 tool in terms of extracting more comprehensive specifications (i.e., ICC values) as well as execution time.

According to Table 8, for apps of ICC-Bench, we observe that VAnDroid2 performs similarly to IC3 in inferring components, intent filters, and intents. The average execution time of VAnDroid2 (1.3 ms) is considerably less than IC3 (5.1 ms). Therefore, for ICC-Bench, VAnDroid2 has been able to extract the same specification from each app compared to IC3. VAnDroid2 significantly outperforms the IC3 tool in terms of execution time. The outputs of running IC3 and VAnDroid2 on Android apps of these two benchmarks are available on the VAnDroid2 website [††].

7.3.2 | Comparing with IccTA, Amandroid, and COVERT

In this section, VAnDroid2 is compared with three existing state-of-the-art static analyzers targeting ICC vulnerability detection: IccTA,[9] Amandroid,[11] and COVERT.[10] All of these analyzer tools have high accuracy in identifying privacy leakage vulnerabilities and are the most related tools to ICC and IAC. Three benchmarks, DroidBench, ICC-Bench, and Ghera, are used in this comparison. The first two benchmarks, DroidBench and ICC-Bench, are two benchmarks of Android applications with ICC-based privacy leaks for which all vulnerabilities are known in advance. DroidBench contains Android apps for evaluating the tools in various static analysis problems, including ICC analysis and inter-app communication analysis. ICC-Bench is another benchmark of Android apps for various purposes, including Intent communication. In this benchmark, each Android app contains at least one ICC leak. As described in Section 6.2, Ghera is a growing repository of Android apps that contains known vulnerabilities in various categories, including ICC. In this comparison, the results of ReproDroid,[16] as a framework to infer the ground truth for data leaks in Android apps, are considered to identify the ground truth for each test case in DroidBench and ICC-Bench. This comparison is based on True Positive (TP), False Positive (FP), and False Negative (FN). Also, the precision, recall, and F-measure of the tools are calculated.

Table 9 presents the results of comparing the effectiveness of the VAnDroid2 tool with IccTA and Amandroid in performing inter-component communication analysis on DroidBench[48] and ICC-Bench.[49] In this comparison, only apps of DroidBench and ICC-Bench that are related to ICC-based privacy leaks are considered. As explained in Section 7.2, a leak is considered as a communication between two components or two apps that originates in one class and ends in another class. Also, a sensitive ICC is a communication from an ICC exit point such as bindService and startActivity to an ICC entry point such as getIntent and onActivityResult. This communication transfers an intent object that can be contained the sensitive data.

As can be seen in Table 9, the results indicate that VAnDroid2 outperforms the other two analysis tools and achieves higher precision (100%), recall (96%), and F-measure (98%) in ICC analysis. VAnDroid2 succeeds in identifying 22 vulnerabilities out of 23 in the DroidBench benchmark and detecting all 18 vulnerabilities in the ICC-Bench suite. In the case of DroidBench, similar to Amandroid, VAnDroid2 has a false negative. The false negative of Amandroid is due to this tool cannot consider the Java Singletone in its modeling. Therefore, Amandroid is not able to detect ICC vulnerability for the *Singletons1* app. The false negative of VAnDroid2 is due to VAnDroid2 uses only static analysis techniques for Android ICC analysis of Android applications. The *UnresolvableIntent1* app has an intent sending mechanism that cannot be resolved statically. Therefore, VAnDroid2 is not able to detect this type of vulnerability.

Among the vulnerability analysis tools (listed in Reference 58), COVERT claimed to detect inter-app vulnerabilities. We compared the effectiveness of VAnDroid2 with COVERT in inter-app vulnerability detection. In this comparison, the extended version of DroidBench proposed by Pauck et al.[16] and the bundles of the Ghera repository are used. Only bundles related to ICC contain Intent Spoofing or Unauthorized Intent Receipt vulnerabilities from the Ghera repository are considered. These bundles are the six bundles introduced in Section 6.2. Table 10 summarizes the results of this

[††] https://mdse.ui.ac.ir/project/vandroid2/

TABLE 9 Comparison of VAnDroid2 with IccTA and Amandroid (TP, FP, and FN are specified by symbols ☑, ☒, ☐, respectively)

(a) DroidBench2.0				(b) ICC-Bench			
Test Case	**IccTA**	**Amandroid**	**VAnDroid2**	**Test Case**	**IccTA**	**Amandroid**	**VAnDroid2**
InterComponentCommunication (ICC)				**Testing ICC Addressing**			
ActivityCommunication1	☑	☑	☑	ICC_Explicit1	☑	☑	☑
ActivityCommunication2	☑☑☒	☑☑☒	☑☑	ICC_Implicit_Action	☑☑	☑☑	☑☑
ActivityCommunication3	☐	☑	☑	ICC_Implicit_Category	☑☑	☑☑	☑☑
ActivityCommunication4	☑☑☒	☑☑	☑☑	ICC_Implicit_Data1	☑☑	☑☑	☑☑
ActivityCommunication5	☑	☑	☑	ICC_Implicit_Data2	☑☑	☑☑	☑☑
ActivityCommunication6	☐	☑	☑	ICC_Implicit_Mix1	☑☑☑	☑☑☑	☑☑☑
ActivityCommunication7	☑	☑	☑	ICC_Implicit_Mix2	☑☑	☑☑	☑☑
ActivityCommunication8	☑☑☒	☑☑	☑☑	ICC_dynregister1	☑☑	☑☑	☑☑
BroadcastTaintAndLeak1	☑☑	☑☑	☑☑	ICC_dynregister2	☑☑☒	☑☑☒	☑☑
ComponentNotInManifest1				Summary			
EventOrdering1	☑	☑	☑	Number of ☑	18	18	18
IntentSink1	☑	☑	☑	Number of ☒	1	1	0
IntentSink2	☑	☑	☑	Number of ☐	0	0	0
IntentSource1	☑	☑	☑	**Precision** P = ☑/(☒+ ☑)	95%	95%	100%
ServiceCommunication1	☐	☑	☑	**Recall** R = ☑/(☐+ ☑)	100%	100%	100%
SharedPreferences1	☑	☑	☑	**F-measure** 2pr /(p + r)	97%	97%	100%
Singletons1	☐	☐	☑				
UnresolvableIntent1	☑☑☑	☑☑☑	☑☑☐				
Summary							
Number of ☑	19	22	22				
Number of ☒	3	1	0				
Number of ☐	4	1	1				
Precision *P* = ☑/(☒+ ☑)	86%	96%	100%				
Recall *R* = ☑/(☐+ ☑)	83%	96%	96%				
F-measure 2pr /(p + r)	85%	96%	98%				

comparison. As can be seen, VAnDroid2 outperforms the COVERT tool and achieves a precision of 100%, a recall of 100%, and an F-measure of 100%, in inter-app analysis.

As shown in Table 10, for the DroidBench benchmark, VAnDroid2 outperforms the COVERT tool and succeeds in detecting all inter-app vulnerabilities. According to the results of applying COVERT to the bundles of Ghera, while COVERT claimed to detect inter-app vulnerabilities, it could not identify any of the ICC vulnerabilities in the six bundles. As can be seen in Table 10, VAnDroid2 significantly outperforms the COVERT tool and detects all the vulnerabilities in these bundles without any further reported security issues. Therefore, according to Tables 9 and 10, VAnDroid2 outperforms the other analysis tools in terms of precision, recall, and F-measure.

7.4 | Other types of ICC vulnerabilities

While Intent Spoofing and Unauthorized Intent Receipt vulnerabilities have been the focus of this paper, we believe that VAnDroid2 can be extended to identify other kinds of ICC vulnerabilities and significant components of VAnDroid2 can be

TABLE 10 Comparison of VAnDroid2 with COVERT (TP, FP, and FN are specified by symbols ☑, ☒, ☐, respectively)

(a) DroidBench3.0				(b) Ghera		
Source App	**Destination App**	**COVERT**	**VAnDroid2**	**Bundle Name**	**COVERT**	**VAnDroid2**
SendSMS	Echoer	☑	☑	DynamicRegBroadcastReceiver	☐	☑
StartActivityForResult1	Echoer	☑	☑	EmptyPendingIntent	☐	☑
DeviceId_Broadcast1	Collector	☑	☑	HighPriority	☐	☑
DeviceId_ContentProvider1	Collector	☑	☑	ImplicitPendingIntent	☐	☑
DeviceId_OrderedIntent1	Collector	☐	☑	StickyBroadcast	☐	☑
DeviceId_Service1	Collector	☑	☑	UnprotectedBroadcastRecv	☐	☑
Location1	Collector	☑	☑	Summary		
Location_Broadcast1	Collector	☑	☑	Number of ☑	0	6
Location_Service1	Collector	☑	☑	Number of ☒	0	0
Summary				Number of ☐	6	0
Number of ☑		8	9	**Precision** $P = ☑/(☒+ ☑)$	0%	100%
Number of ☒		0	0	**Recall** $R = ☑/(☐+ ☑)$	0%	100%
Number of ☐		1	0	**F-measure** 2pr /(p + r)	0%	100%
Precision$P = ☑/(☒+ ☑)$		100%	100%			
Recall$R = ☑/(☐+ ☑)$		89%	100%			
F-measure 2pr /(p + r)		94%	100%			

reused. Each further security analysis is built on top of VAnDroid2 and involves extending two phases: the Transformation and Integration phase and the Analysis phase (i.e., phases 1 and 2 in Figure 7).

As described in Section 2.2, the ICC mechanism can be done through URIs. URIs are used to communicate with the Content Provider component as a database for an Android app. We still see various reports about security issues in Android apps, including insecure interprocess communication and data leakage problems.[59,60] Therefore, an important category of inter-app vulnerabilities is information data leakage. For this type of vulnerability, VAnDroid2 must be extended to consider ICC information related to URIs and data sharing as the main mechanism for Android that allows app components to share data.

To show the extension potential and reuse of VAnDroid2, we describe an example of inter-app information leakage vulnerability. Consider a bundle of the Ghera repository called *InadequatePathPermission-InformationExposure*. This bundle contains benign and malicious apps. The benign app has a Provider called *UserDetailsContentProvider* that is exported (i.e., the other app components can access this provider). This app component has a path-permission *edu.ksu.cs.benign.permission.internalRead* with the path prefix */user*. This permission controls access to a folder and has no control access over subfolders. The malicious app has a *MalActivity* component. This malicious component has created a read operation request for data within the provider of benign. This data read request has a URI *content://edu.ksu.cs.benign.userdetails/user/ssn*.

As explained above, the provider of benign protects the data within the */user* folder and does not perform any protection for the data within the subfolders */user/ssn*. Therefore, the malicious app can exploit this ICC vulnerability to perform malicious activities.

The following efforts are required to extend VAnDroid2 to support the analysis of this ICC attack scenario.

- **Extend the transformation and integration phase**. First, the Android Application Security Aspects metamodel must be extended to address the concepts related to the URIs and data sharing mechanism. Since the Provider is responsible for sharing data between app components, it has a complex security model.[29] As depicted in Figure 19, according to the metamodel (the part related to the *ContentProvider* element), *ContentProvider* needs to be extended to address all scopes of provider permissions such as path-permission. Second, the ICC metamodel must be extended to extract

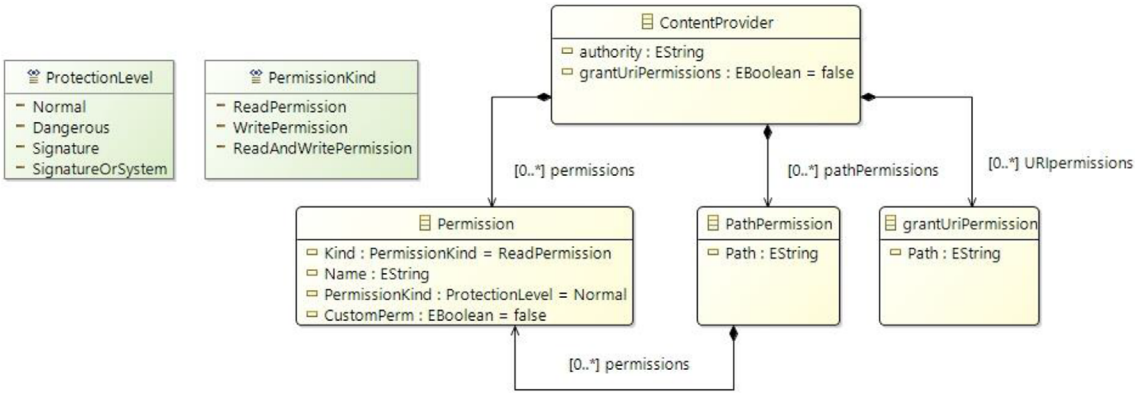

FIGURE 19 The extended part related to the *ContentProvider* element of Figure 5

all potential data sharing interactions between app components at intra- and inter-app levels. Finally, the M2M transformations need to be extended to extract all ICC information from bundles of Android apps, which is needed for analyzing data sharing communications between app components.

- **Extend the analysis phase**. The formal analysis process needs to be extended to identify the data leakage vulnerabilities at intra- and inter-app communication levels.

7.5 | Comparing with the original VAnDroid framework

Tables 11 and 12 present a comparison between VAnDroid2 and the original VAnDroid framework[19] according to the following dimensions.

- Approach Positioning (Research Problem). This dimension (the first part of Table 11) characterizes the objectives of VAnDroid and VAnDroid2.
- Approach. The second part of Table 11 characterizes the approaches presented in VAnDroid and VAnDroid2.
- Tool support. The third part of Table 11 specifies the features of two tools, VAnDroid and VAnDroid2.
- Evaluation and results. The first part of Table 12 is about the evaluation of VAnDroid and VAnDroid2.
- Comparing with state-of-the-art analysis tools. The second part of Table 12 is about the comparison VAnDroid and VAnDroid2 with state-of-the-art tools.

7.6 | Limitations and threats to validity

7.6.1 | Limitations

One limitation of VAnDroid2 is related to input Android applications. Since VAnDroid2 uses MoDisco to generate the Java model, only apps with standard GUI widgets for Android API and without anonymous classes can currently be considered. To overcome this limitation, we can improve MoDisco or consider another tool to generate the Java model. Another limitation is related to the analysis of "strings" of intent values. Currently, VAnDroid2 can only perform simplified string analysis to extract the intent values. Nevertheless, VAnDroid2 can be extended to perform more complex string analyses of Android applications.

7.6.2 | Threats to validity

One threat to the validity of our evaluation results is the generalization of obtained results to bundles outside of our study. To overcome this threat, for 10 experiments, we considered a combination of benign apps, malicious apps, and vulnerable

TABLE 11 Comparison of VAnDroid2 with the original VAnDroid framework

	VAnDroid[19]	VAnDroid2
Approach positioning (research problem)	Focus on analyzing a single app component in isolation to identify intent-based security issues at the intra-component analysis level.	Focus on analyzing multiple app components to identify intent-based security issues at four levels of analysis: (1) intra-component, (2) inter-component, (3) intra-app, and (4) inter-app
Approach	Propose an automated and model-based approach to conduct an MDRE process for supporting the intra-component analysis.	Propose an automated, incremental, compositional, and MDRE approach to analyze ICCs at four analysis levels.
	VAnDroid contains three phases:	VAnDroid2 contains three phases:
	Model discovery:	**Model Discovery**:
	Present a model-based static program analysis for a single Android app to extract the IR from the app.	Present a model-based static program analysis for multiple Android apps to extract the IR from apps.
	Transformation and integration:	**Transformation and integration**:
	Present the Android application security aspects metamodel to extract the high-level model from the app.	1. Present an expanded version of Android Application Security Aspects Metamodel to extract the high-level model from each app in a bundle.
		2. Present a brand new metamodel called ICC to identify all intent-based ICCs at both intra and inter-app levels.
		3. Present a model-based ICC extractor for Android at intra and inter-app levels.
		4. Implement an algorithm for model-based app component analysis to conduct a model-based precise intent resolution process.
	Analysis:	**Analysis**:
	Present formal model-based analysis processes for Intent Spoofing and Unauthorized Intent Receipt detection at one analysis level:	Present formal model-based analysis processes for Intent Spoofing and Unauthorized Intent Receipt detection at four analysis levels:
	- intra-component	intra-component
		- inter-component
		- intra-app
		- inter-app
Tool support	Propose an Eclipse-based tool to receive a single app and analyze a single component in isolation to identify intent-based security issues.	Propose an Eclipse-based tool to receive multiple apps and perform incremental ICC analysis at four analysis levels:
		- intra-component
		- inter-component
		- intra-app
		- inter-app

TABLE 12 Comparison of VAnDroid2 with the original VAnDroid framework (Cont.)

	VAnDroid[19]	VAnDroid2
Evaluation and results	**Correctness**	**Correctness**:
	1. Consider four apps evaluated by Peck and Northern[61]	1. Consider the six bundles of the Ghera repository
	2. Apply the proposed tool to these apps	2. Apply VAnDroid2 to these six app bundles
		3. Analyze the correctness of the analysis results
		4. Compare the results of VAnDroid2 with the results of COVERT and DIALDroid
	Results: VAnDroid's analysis results are reliable, and this tool can correctly reveal the intra-component issues in a single Android app.	Results: VAnDroid2's analysis results are reliable, and this tool can correctly reveal the inter-app ICC attacks in Android app bundles.
	Scalability:	**Scalability**:
	1. Apply VAnDroid to 20 apps from Google Play and 110 apps from the F-Droid repository	1. Construct a dataset of benign, malicious, and vulnerable apps from various repositories
	2. Evaluate the scalability of the analysis results according to deal with the Large apps (code size) issue	2. Create 10 bundles of apps, each containing 35 apps, from the provided dataset
		3. Apply the VAnDroid2 to these app bundles
		4. Evaluate the scalability of the analysis results according to deal with the several issues (issues introduced in Section 6.3)
	Results: VAnDroid has supported the scalability criterion (according to the code size) in performing intra-component analysis of real-world Android applications.	Results: VAnDroid2 has supported the scalability criterion in performing inter-app ICC vulnerability analysis.
	Usability:	**Run-Time Performance**:
	The ability of VAnDroid in analyzing real-world Android apps and detecting their vulnerabilities at the intra-component analysis level is analyzed.	1. Without considering incremental ICC analysis: this tool is able to analyze app bundles containing hundreds of components in a few minutes.
		2. With considering incremental ICC analysis: the effects of VAnDroid2's incremental ICC analysis capability are visible when the size of the app bundles increases.
Comparing with the state-of-the-art analysis tools	Compare with IccTA, DroidGuard, FlowDroid, and Amandroid:	1. Compare with IC3:
	VAnDroid significantly outperforms compared with four tools in performing intra-comment analysis.	VAnDroid2 significantly outperforms in terms of extracting more comprehensive specifications as well as execution time.
		2. Compare with IccTA and Amandroid:
		VAnDroid2 significantly outperforms the other tools and achieves higher precision, recall, and F-measure at the intra-app analysis level.
		3. Compare with COVERT:
		VAnDroid2 significantly outperforms COVERT and reaches a precision of 100%, a recall of 100%, and an F-measure of 100% at inter-app level.

apps. For benign and malicious apps, AndroZoo,[47] as a growing collection of apps from various popular marketplaces is considered. For vulnerable apps, all known vulnerability benchmarks are used.

Another threat is related to the results of comparing the ability of VAnDroid2 in identifying ICC vulnerabilities with other state-of-the-art tools. In this comparison, the reported accuracy of VAnDroid2, in terms of precision, recall, and F-measure, depends on the quality of the dataset that we use as a ground truth with known security attacks. To overcome this threat, we used the results of ReproDroid,[16] as a framework to identify the ground truth for data leaks of each test case in DroidBench and ICC-Bench. In the results of the ReproDroid framework, true positives and false positives are identified for each test case. In fact, an ICC communication has been identified that can cause a security issue (i.e., ICC-based privacy leakage).

8 | CONCLUSION

In this paper, a framework called VAnDroid2 was presented, as an extension to our previous work, to improve the detection of IAC security issues. VAnDroid2, based on MDRE, has three phases. At first, the comprehensive IR of each Android application is created without losing information. Then, by collecting security information in the form of domain-specific models from each app, the comprehension of the Android system is facilitated and all potential intent-based communications are extracted from a bundle of Android applications. Finally, by paying a one-time cost of modeling the security structure and specification of Android apps, the inter-app security analysis is performed and the results are displayed to the user using XMI models. VAnDroid2 is developed as an Eclipse-based tool. This tool has been applied to hundreds of real-world Android apps, and the correctness, scalability, and run-time performance are examined. VAnDroid2 is also compared with several existing state-of-the-art tools related to inter-component communication and inter-app communication analysis of Android apps. The evaluation results indicate that VAnDroid2 has been able to conduct a more effective IAC security analysis and achieve higher precision, recall, and F-measure in inter-app vulnerability detection.

To create the bundles of real-world apps, we randomly selected the apps from the created dataset. Since these bundles may not reflect the real bundles of installed Android applications on the users' devices, as first future work, the information provided by marketplaces, such as Bazaar, for each Android app can be considered. Bazaar, as a local app store, provides the *Users Also Installed* section on the page of each Android app in the store. Based on this information, we can specify the apps that exist together on the users' devices. Violation of the least-privilege principle in Android can cause serious security issues in Android ICC, including privilege escalation attacks. Hence, as second future work, the proposed approach can be expanded to detect vulnerabilities that lead to privilege escalation attacks, as one of the major categories of ICC vulnerabilities. Since VAnDroid2 focuses just on ICC analysis, as third future work, the proposed approach can be developed to consider URIs to detect other inter-app vulnerabilities such as passive data leaks and content pollution.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in MDSE research group website at https://mdse.ui.ac.ir/project/vandroid2/.

ORCID

Bahman Zamani https://orcid.org/0000-0001-6424-1442
Behrouz Tork-Ladani https://orcid.org/0000-0003-2280-8839
Jacques Klein https://orcid.org/0000-0003-4052-475X

REFERENCES

1. Ranganath VP, Mitra J. Are free android app security analysis tools effective in detecting known vulnerabilities? *Empir Softw Eng*. 2020;25(1):178-219. doi:10.1007/s10664-019-09749-y
2. Hurier M. *Creating Better Ground Truth to Further Understand Android Malware: A Large Scale Mining Approach Based on Antivirus Labels and Malicious Artifacts*. PhD thesis, University of Luxembourg, Luxembourg; 2019.
3. Statista. Google play store: number of apps 2020; July 30; 2021. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
4. Hammad M. *Self-Protection of Android Systems from Inter-Component Communication Attacks*. PhD thesis, University of California, Irvine; 2018.
5. Sadeghi A. *Efficient Permission-Aware Analysis of Android Apps. PhD thesis*. University of California, Irvine; 2017.

6. Li L, Bissyandé TF, Papadakis M, et al. Static analysis of android apps: a systematic literature review. *Inf Softw Technol*. 2017;88:67-95. doi:10.1016/j.infsof.2017.04.001
7. Octeau D, McDaniel P, Jha S, et al. Effective inter-component communication mapping in android: an essential step towards holistic security analysis. Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13); 2013; Washington, DC; LA-UR-13-26794.
8. Arzt S, Rasthofer S, Fritz C, et al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Not*. 2014;49(6):259-269. doi:10.1145/2666356.2594299
9. Li L, Bartel A, Bissyandé TF, et al. IccTA: detecting inter-component privacy leaks in android apps. Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering; 2015.
10. Bagheri H, Sadeghi A, Garcia J, Malek S. COVERT: compositional analysis of android inter-app permission leakage. *IEEE Trans Softw Eng*. 2015;41(9):866-886. doi:10.1109/TSE.2015.2419611
11. Wei F, Roy S, Ou X. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans Priv Secur (TOPS)*. 2018;21(3):1-32. doi:10.1145/3183575
12. Wu T, Deng X, Yan J, Zhang J. Analyses for specific defects in android applications: a survey. *Front Comput Sci*. 2019;13:1210-1227. doi:10.1007/s11704-018-7008-1
13. Sadeghi A, Bagheri H, Garcia J, Malek S. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Trans Softw Eng*. 2016;43(6):492-530. doi:10.1109/TSE.2016.2615307
14. Bagheri H, Wang J, Aerts J, Malek S. Efficient, evolutionary security analysis of interacting android apps. Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME); 2018.
15. Reaves B, Bowers J, Gorski SA III, et al. Android: assessment and evaluation of android application analysis tools. *ACM Comput Surv (CSUR)*. 2016;49(3):1-30. doi:10.1145/2996358
16. Pauck F, Bodden E, Wehrheim H. Do android taint analysis tools keep their promises? Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2018.
17. Bruneliere H. *Generic Model-based Approaches for Software Reverse Engineering and Comprehension*. PhD thesis. Nantes University; 2018.
18. Sabir U, Azam F, Haq SU, Anwar MW, Butt WH, Amjad A. A model driven reverse engineering framework for generating high level UML models from java source code. *IEEE Access*. 2019;7:158931-158950. doi:10.1109/ACCESS.2019.2950884
19. Nirumand A, Zamani B, Tork LB. VAnDroid: a framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique. *Softw Pract Exp*. 2019;49(1):70-99. doi:10.1002/spe.2643
20. Octeau D, Luchaup D, Dering M, Jha S, McDaniel P. Composite constant propagation: application to android inter-component communication analysis. Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering; 2015.
21. Statista. Mobile OS market share 2021. Accessed July 30, 2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
22. Bhat P, Dutta K. A survey on various threats and current state of security in android platform. *ACM Comput Surv*. 2019;52(1):1-35. doi:10.1145/3301285
23. Chin E, Felt AP, Greenwood K, Wagner D. Analyzing inter-application communication in Android. Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services; 2011:239-252.
24. Ma C, Wang T, Shen L, Liang D, Chen S, You D. Communication-based attacks detection in android applications. *Tsinghua Sci Technol*. 2019;24(5):596-614. doi:10.26599/TST.2018.9010133
25. Samhi J, Bartel A, Bissyandé TF, Klein J. RAICC: revealing atypical inter-component communication in android apps. Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE); 2021.
26. Android Developers. Context. Accessed July 3, 2021. https://developer.android.com/reference/android/content/Context
27. Android Developers. Pendingintent. Accessed July 3, 2021. https://developer.android.com/reference/android/app/PendingIntent
28. Android Developers. IntentSender. Accessed July 30, 2021. https://developer.android.com/reference/android/content/IntentSender
29. Six J. *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*. O'Reilly Media, Inc; 2011.
30. Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: a model transformation tool. *Sci Comput Program*. 2008;72(1-2):31-39. doi:10.1016/j.scico.2007.08.002
31. Eclipse Foundation. Eclipse Acceleo project. Accessed July 30, 2021. https://www.eclipse.org/acceleo/
32. Bruneliere H, Cabot J, Dupé G, Madiot F. MoDisco: a model driven reverse engineering framework. *Inf Softw Technol*. 2014;56(8):1012-1032. doi:10.1016/j.infsof.2014.04.007
33. Brambilla M, Cabot J, Wimmer M. Model-driven software engineering in practice. Synthesis lectures on software engineering; 2017.
34. Raibulet C, Fontana FA, Zanoni M. Model-driven reverse engineering approaches: a systematic literature review. *IEEE Access*. 2017;5:14516-14542. doi:10.1109/ACCESS.2017.2733518
35. Lu L, Li Z, Wu Z, Lee W, Jiang G. CHEX: statically vetting Android apps for component hijacking vulnerabilities. Proceedings of the 2012 ACM Conference on Computer and Communications Security; 2012:229-240.
36. Octeau D, Jha S, McDaniel P. Retargeting Android applications to Java bytecode. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering; 2012:1-11.
37. Gordon MI, Kim D, Perkins JH, Gilham L, Nguyen N, Rinard MC. Information-flow analysis of android applications in DroidSafe. Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS); 2015.
38. Tiwari A, Groß S, Hammer C. IIFA: modular inter-app intent information flow analysis of android applications. Proceedings of the International Conference on Security and Privacy in Communication Systems; 2019:335-349.

39. Klieber W, Flynn L, Bhosale A, Jia L, Bauer L. Android taint flow analysis for app sets. Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis; 2014:1-6.
40. Jha AK, Lee S, Lee WJ. Modeling and test case generation of inter-component communication in android. Proceedings of the 2015 2nd ACM International Conference on Mobile Software Engineering and Systems; 2015:113-116.
41. Biswas S, Sharif K, Li F, Liu Y. 3P framework: customizable permission architecture for mobile applications. Proceedings of the International Conference on Wireless Algorithms, Systems, and Applications; 2017:445-456.
42. Biswas S, Haipeng W, Rashid J. Android permissions management at app installing. *Int J Sec Appl*. 2016;10(3):223-232. doi:10.14257/ijsia.2016.10.3.21
43. Hammad M, Bagheri H, Malek S. DelDroid: an automated approach for determination and enforcement of least-privilege architecture in android. *J Syst Softw*. 2019;149:83-100. doi:10.1016/j.jss.2018.11.049
44. Github. Jadx: Dex to Java decompiler. Accessed July 30,2021. https://github.com/skylot/jadx
45. Android Developers. Intent and intent filters. Accessed July 30, 2021. https://developer.androidcom/guide/components/intents-filters
46. Nirumand A, Zamani B, Tork L, et al. ATL rules and OCL queries implemented in VAnDroid2. Technical report, MDSE Research Group; 2022. https://mdse.ui.ac.ir/TR/UI-SE-MDSERG-2022-05.pdf.
47. Allix K, Bissyandé TF, Klein J, Le Traon Y. AndroZoo: collecting millions of android apps for the research community. Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR); 2016:468-471.
48. GitHub. Secure-software-engineering/DroidBench. Accessed July 30, 2021. https://github.com/secure-software-engineering/DroidBench
49. GitHub. fgwei/ICC-Bench. Date Accessed: July 30, 2021. fgwei/ICC-Bench.
50. Qiu L, Wang Y, Rubin J. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2018:176-186.
51. Statistics. 55+ jaw dropping app usage statistics in 2021. Accessed July 30, 2021. https://techjury.net/blog/app-usage-statistics/
52. Statistics. Mobile app download and usage statistics (2021); Accessed July 30, 2021. https://buildfire.com/app-statistics/
53. Statistics. Most popular Google play app categories as of 1st quarter 2022. Accessed May 12, 2022. https://www.statista.com/statistics/279286/google-play-android-app-categories/
54. Pressman RS, Maxim BR. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education; 2015.
55. Bosu A, Liu F, Yao D, Wang G. Collusive data leak and more: large-scale threat analysis of inter-app communications. Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security; 2017:71-85.
56. Bondi AB. Characteristics of scalability and their impact on performance. Proceedings of the 2nd International Workshop on Software and Performance; 2000:195-203.
57. Clements PC. *Software Architecture in Practice*. Dissertation. Software Engineering Institute; 2002.
58. Bitbucket. Android-app-vulnerability-benchmarks. Accessed February 10, 2021. https://bitbucket.org/secure-it-i/may2018/src/master/vulevals/
59. What are the most critical android application vulnerabilities of 2021? Accessed May 14, 2022. https://www.hackingloops.com/most-critical-android-application-vulnerabilities/
60. OWASP. OWASP Top 10 – 2021. Accessed May 14, 2022. https://owasp.org/Top10/
61. Peck M, Northern C. Analyzing the effectiveness of app vetting tools in the enterprise. MITRE Corporation, Technical Report; 2016.

How to cite this article: Nirumand A, Zamani B, Tork-Ladani B, Klein J, Bissyandé TF. A model-based framework for inter-app Vulnerability analysis of Android applications. *Softw Pract Exper*. 2022;1-42. doi: 10.1002/spe.3171