Khan *et al. Hum. Cent. Comput. Inf. Sci.* (2018) 8:21
https://doi.org/10.1186/s13673-018-0144-7

Human-centric Computing and Information Sciences

RESEARCH Open Access

CrashSafe: a formal model for proving crash-safety of Android applications

Wilayat Khan[1*], Habib Ullah[2], Aakash Ahmad[2], Khalid Sultan[2], Abdullah J. Alzahrani[2], Sultan Daud Khan[2], Mohammad Alhumaid[2] and Sultan Abdulaziz[2]

*Correspondence:
wilayat@ciitwah.edu.pk
[1] COMSATS University, Islamabad, Wah Campus, Wah Cantt, Pakistan
Full list of author information is available at the end of the article

Abstract

Each software application running on Android powered devices consists of application components that communicate with each other to support application's functionality for enhanced user experience of mobile computing. Application components inside Android system communicate with each other using inter-component communication mechanism based on messages called intents. An android application crashes if it invokes an intent that can not be received by (or resolved to) any application on the device. Application crashes represent a severe fault that relates to compromised users' experience, consequently resulting in decreased ratings, usage trends and revenues for such applications. To address this issue—by formally proving crash-safety property of Android applications—we have defined a formal model of Android inter-component communication using Coq theorem prover. The mathematical model defined in theorem prover allows one to prove the properties of inter-component communication system and check the correctness of the proof in an automated way. To demonstrate the significance of the formal model developed, we carried proof of crash-safety of Android applications using Coq tool. The proposed solution named CrashSafe supports a formal approach that enables one to (i) check the correctness of inter-component communication in Android systems and (ii) establish a formal foundation for other tools to assess Android applications' reliability and safety.

Keywords: Android, Inter-component communication, Mobile computing, Coq, Formal analysis, Security and reliability

Introduction

Mobile devices have become an indispensable part of modern life style. Originally designed and built to facilitate remote communications such as phone calls and text messaging, mobile devices now support portable computing, context-aware communication, enhanced user interaction, and high-connectivity systems [36]. The operating system—that powers-up mobile devices—enables the execution of third party applications (apps for short) that support a variety of tasks on the go [21]. These capabilities of modern mobile devices make them *smart* and open up the gate for a multitude of applications that support a variety of tasks that includes but not limited to mobile commerce, health monitoring, and location querying [12]. There are thousands of mobile

Springer Open

© The Author(s) 2018. This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

applications available on application stores such as Google Play and Apple Apps Store for the two most popular mobile operating systems Android and iOS [2, 16].

Android is Google's open source applications development platform as well as an operating system. It is based on Linux kernel and powers up mobile devices such as smart phones and tablets [17]. According to Statista [39], Android's market share in sales to end users in second quarter of 2017 was 87%. Android provides a software development kit (SDK) with other developer tools and application programming interface (API) for building innovative mobile applications. Application components, namely *activity*, *service*, *broadcast receiver* and *content provider* are the essential building blocks of an Android application. Such a modular application framework allows mutually non-trusting Android applications to share their functionalities and resources using security enforcement mechanism based on permissions [15, 18]. These applications can access (hardware and software) resources of the device, such as camera and contacts, and share data with other applications or components of the same application. In Android, messaging or data passing between applications on a mobile device is achieved with inter-component communication (ICC) using *intents* [28]. Through the ICC messaging, both control and data can flow between applications that may lead to numerous security and reliability issues such as a malicious application trying to access private data or application crashes that may lead to catastrophic impacts on mobile computing. If an application, for example, crashes due to failure of intent resolution, the user is likely to abandon the application for a competitor [30].

Considering the security perspective of Android framework, ICC can be exploited by the attackers to leak sensitive user information. A recent case for compromise on data security and privacy is reported in [27], where a gaming application frequently monitored the viewing habits of their users (some of them may be children) even when the game was not active. By using a device's microphone, the gaming application was able to sense what people watched by identifying audio signals in TV ads and shows. Even worst, the gaming application could match the information about the places people visit and the movies they watch. Second, from the application's stability point of view, application crashes negatively impact the performance and user experience of mobile computing. This means that, Android's ICC that enables data communication between components within or outside application can also compromise the data security, privacy and stability of mobile applications [13]. The security aspect of applications is equally important, though, the main focus of this paper is to study applications' safety against crashes caused by failure in intent resolution.

Research context and challenges Android applications can crash at run time due to intents, i.e., an application crashes if there is no application component that can receive the intent [20]. In an empirical study, Maji et al. tested more than 6 million intents against more than 800 applications and found that around 10% Android components tested crashed due to intents. In a separate analysis, Chin et al. [9] found 1414 exposed surfaces in 100 Android applications, with most of them (1013) were caused by intents. Application crash is the major problem (62% of all) faced by the users, with most of them drive users to uninstall the app [41]. ICC, in addition, can cause information leak: Li et al. [26] tested 15,000 Google Play applications and found that about 16% (337 out of 15,000) were leaking information through ICC.

The existing mechanism to protect against such crashes and information leaks is to check explicitly, before invoking an intent, that the intent follow certain security rules: e.g. checking there exists at least an application component to address the intent. To the best of our knowledge, none of these approaches provide formal proofs of guarantee that security rules are indeed followed. Android applications' security and reliability has been studied from different perspective, including structure of applications [38], permission systems [8, 15] and ICC [7, 8, 32, 40]. These analysis are helpful to investigate and mitigate the security challenges that relate to mobile applications in general but they lack security and reliability analysis targeted towards ICC and in particular intent resolution. In the context of Android applications, one of the most vulnerable part of Android framework is its ICC [7, 40] that require a rigorous and formal analysis to ensure application's safety, reliability and security.

The official documentation [19] of Android's ICC is in English—an informal documentation—that leads to misinterpretations, and lack of formal reasoning and analysis. The lack of formal reasoning of informal documentation and methods affects their overall acceptance. The digital forensic used in criminal investigation and evidence, for example, is unproven and is highly criticised in legal proceedings [4]. Such non-rigorous specifications can cause interpretation issues, most commonly developers often choosing undocumented practices that can lead to applications vulnerable to security and privacy threats [1]. A typical consequence of misinterpreting the informal documentation may lead to a scenario where an application developer may incautiously expose a component to third party applications [23]. Furthermore, as the informal specification of ICC do not have a mathematical foundation and hence cannot be used to formally prove correctness of Android applications.

Solution overview In this paper, we propose to formally specify the ICC—as defined in Android's official documentation—using theorem prover Coq [22]. The formalism is a simple but careful translation from English to logical notations and formulas. An overview of the proposed solution is provided in Fig. 1 that illustrates formal specification of the Android application's ICC from its informal documentation. Android application components and their interaction with each other through ICC is currently viewed through its English documentation. The informal (English based) documentation for ICC can be intuitive and simple for application developers but it can not be used for formal reasoning. The informal specification of ICC is manually translated to a precise formal Coq specification that is machine readable. Once a formal specification defined in the logic of Coq is available, the Coq proof facility can be used to formally prove theorems about ICC. This is demonstrated by proving the correctness of Android ICC using the Coq proof facility. Furthermore, as the tools for security analysis normally rely on formal models [29], the formal model in Coq can provide a foundation for such tools for a rigorous and automated analysis of ICC. The major contributions of this paper are:

- To enable a formal reasoning of Android's ICC using computer tools, a formal model of the Android's ICCs is built using the logic behind Coq theorem prover,
- To demonstrate the effectiveness of our formal model, a simple Android application is encoded using the formal notations developed, and

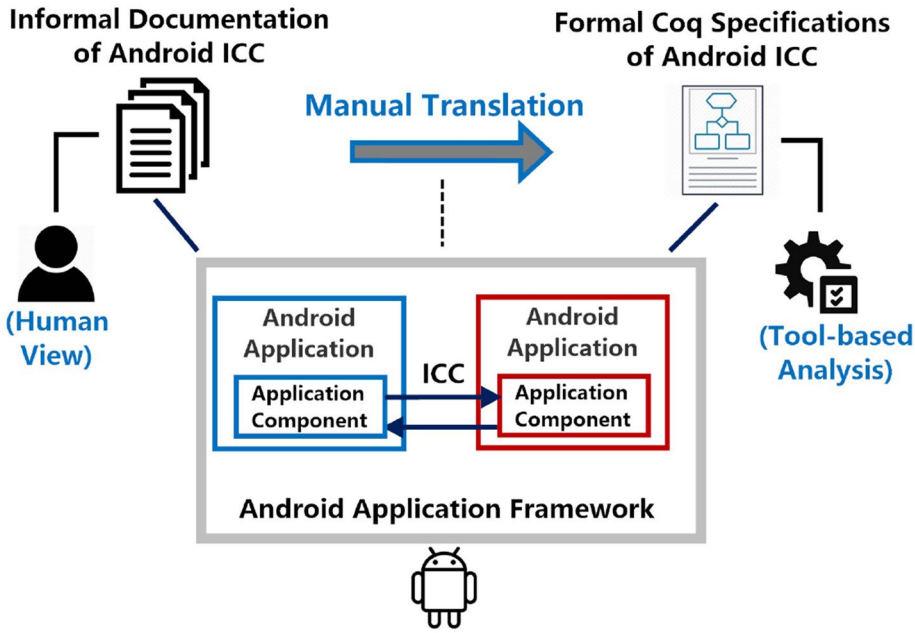

Fig. 1 Overview of the proposed solution in the context of Android ICC

- To support proofs, formal proofs of crash-safety property in general and in particular for the encoded application are conducted using the proof facility of Coq.

The proposed solution CrashSafe offers twofold benefits including formal (i) proof of correctness of inter-component communication in Android systems and (ii) foundations for other tools to assess Android applications' reliability and safety. The rest of the paper is organized as the following: In the next section, a brief overview of the Android application framework and ICC is given. In "Formalizing inter-component communication" section, the Coq formalization of the ICC and encoding of a simple application in the formal notations developed is presented. The proofs of correctness of Android ICC are listed in "Proofs" section. A summary of the related work is presented in "Related work" section and the paper is concluded in "Conclusion and future research" section.

Background

In this section, we present the background information and technical details about (i) Android application framework and its components, (ii) inter component communications along with (iii) types and (iv) structure of the intent. The concepts and terminologies introduced in this section are used throughout the paper.

Android application framework and components

Android application framework allows developers to build innovative applications for mobile devices using programming languages Koltlin, C++ and Java [18]. The Android SDK tools compile the application source code into an Android application package (APK) containing all the contents of Android application which is then used by Android-powered devices to install the application. Android operating system treats each

application as a different user and assigns each of them a unique user ID. Each application is executed in isolation by a separate process using its own virtual machine. By default, Android implements the principle of *least privilege*, where each application is given permission only to access components required for its work and no more [17]. Applications with the same user ID (e.g., by sharing the same Linux user ID) can share data with each other and can run in the same process.

In Android systems, applications comprise of a number of components that can be classified into four main types namely: *activity*, *service*, *receiver* and *provider*. Each of these components provides a different entry point for the Android framework to manage the applications [9].

- *Activity* is the entry-point for interacting with the user of the device through a single screen. It supports different types of activities performed in the Android application framework. Typical example of activity component are users' activities such as interaction with the system, input to the system or manipulation of application's logic and data. Specifically, user interface that allows application display and enables user interaction with it is managed by the activity component. A single application can have more than one activity and the user can switch between different activities.
- *Service* component is a general-purpose entry point and supports variety of (background or foreground) services such as audio and visual notifications, component authentication, or application execution monitoring. Services may run in the background and perform long-running operations without necessarily providing any user interface. Alternatively, services may interact with users and notify them through service notifications. For example, an anti-spam application can continuously execute in the background and can update the user only when a potential spam is detected.
- *Broadcast receiver* is the type of components that depends on other components to receive any activity or service to complete their functionality. For example, the broadcast receivers receive intents from Android application framework. Intents [19] are type of messages used by applications to request functionalities from other services or activities. Most of the broadcast messages originate from the system, for example, the system may announce that the battery is low. Applications may also initiate broadcasts to let other applications know about a broadcast event (e.g., when some data is downloaded and is available for them to use).
- *Content provider* provides services or required functionality to other components through component communication. In other words, the content providers provide data storage to the applications. Other applications can access the data by querying the storage or even can modify it provided that the content provider allows it.

Intents

Android is a Linux-based operating system where each application is assigned a unique user ID and is run in an isolated virtual machine to provide better application management and security [37]. To work as a single entity for better user experience, Android applications uses intents, mediated by Android runtime, to share data and services with each other. This sharing can be inter-application (a component of one application

communicate with a component of another) or intra-application (among components within an application). One component sends an intent including optional name of target component, name of action to be performed, data to operate on and category. The receiving component, likewise, include *intent filter* with specification of intents that it is interested to receive.

An *action* is a string that specifies the action to be performed (e.g., *VIEW*) or has happened and is reported (e.g., *BATTERY_LOW*). In addition, intent includes a (possibly empty) list of *categories*, each contains additional information about the component that should handle the intent. The *data* field of the intent include a uniform resource identifier (URI) to identify the data to act upon and the multi-purpose internet mail extensions (MIME) data type. The type of the data can be inferred from the data itself, though, it is important to add the type of data as it helps Android system to locate the most appropriate component for the intent. The intent fields just described are sufficient for the system to identify a relevant component it should start to receive the intent and hence are included in the formal definitions ("Formalizing inter-component communication" section). There are additional fields to carry *extra* information as key-value pairs and *flags* to carry metadata, however, none of them affects the way an intent is resolved to a component.

Based on the way a set of target components are identified, intents are categorized as *explicit* and *implicit*. In an explicit intent, a component is addressed explicitly using a fully-qualified name and are normally used in an application to communicate with its own components as they are known to the owner. For explicit intent, the mechanism to find an appropriate component for an action is straight forward: the component is described by a full-qualified name inside the intent which the system can easily locate. On the other hand, in an implicit intent, a component is addressed using other fields action, data (URI and MIME type) and category. As a running example to understand intents and intent filters, consider an *Email* and a *Browser* application are installed on Android device. The *Email* application displays an email message, having a hyperlink to a web page, in an activity. When the user clicks on the link in the message displayed to the user by the activity of the *Email* application, the activity sends an intent to Android system for opening (viewing) the requested web page by the browser.

An implicit intent `exampleIntent` is created as an object of `Intent` class using the `new` constructor as shown in Listing 1. The target component name is not provided anywhere which makes the intent created implicit. The values for the fields action, category, data URI and data (MIME) type are added using methods `setAction()`, `addCategory()`, `setData()` and `setType()`, respectively on lines 3–7. The next function `startActivity()` on line 8 starts an activity for the desired action and provides the intent `exampleIntent` including other necessary data. The implicit intent in Listing 1 can be made explicit just by replacing the code on line 2 with `Intent exampleIntent = new Intent(this, exampleActivity.class));`. This code adds the name of target activity `exampleActivity` explicitly to the intent created.

```
String URL = "https://www_example.com:200/intents.html";
Intent exampleIntent = new Intent();
 exampleIntent.setAction(Intent.ACTION_VIEW);
 exampleIntent.addCategory(Intent.CATEGRY_DEFAULT);
 exampleIntent.addCategory(Intent.CATEGRY_BROWSABLE)
 exampleIntent.setData(Uri.parse(URL));
 exampleIntent.setType("text/html");
 startActivity(exampleIntent);
```

Listing 1 Example of an intent

Inter-component communication

For an application to communicate with another application, a component of the first application creates and sends an intent to a component of the target application, as shown in Fig. 2. The component A wants component B to perform some *action* on its *data* of a particular *category*. To do this, it creates an intent with name of action, data and its category and sends it to B, which performs the requested action on the data.

Communication based on intents can be with or without the target component name in intent. If a target name is included in the intent (*explicit* intent), Android system passes on the intent to the target component. Inter-component communication using explicit intent is straight forward and is not included in this work. To address a component using an intent without the target component name (*implicit* intent), a general action is declared and the target is resolved by the Android system (runtime). The type of action in implicit intent, to be performed by a component, enables the system to find a set of appropriate components for that action. Finding the target component for an implicit intent is complicated: the system takes the intent's data and looks for an appropriate set of components in the applications available on the mobile device. In case only one appropriate component is found, the system start that component, however, for more options, the system asks the user using a dialogue to choose from the list.

When a component intends to start another component using implicit intent (see example below), it sends an intent and the system looks for appropriate components by looking into the intent content (action) and comparing with intent filters declared in applications on the device. Android enforces this mechanism using *intent filters* (Listing 2) declared in a manifest file: application specifies the type of intents in its manifest file it should receive. For the *Browser* application to receive the intent created by the *Email* application (Listing 1), the *Browser* must have the filter as defined in Listing 2. This filter declares an action `ACTION_VIEW`, categories `DEFAULT` and `BROWSABLE` and data URL and type `text/hmtl` for an activity to display the web page in a tab. When the *Browser* application receives this intent, it is checked against the filter and as it can deliver the requested service (see proof of this in "Proofs" section), an activity of the *Browser* opens the web page from the URL provided in a tab. Declaring an intent filter for a component

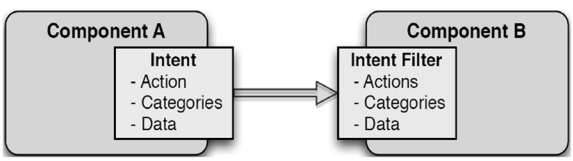

Fig. 2 High-level description of inter-component communication [32]

enable other applications to start it. A component, with no intent filter declared in manifest file, can only be started by an explicit intent.

```
<intent-filter>
    <action android:name="android.intent.action.ACTION_VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:mimeType="text/html"
          android:scheme="https"
          android:host="www.example.com"
          android:port="200"
          android:path="intents.html" />
    ...
</intent-filter>
```

Listing 2 Example of an intent filter

The inter-component communication between two components in Fig. 2 appears a direct communication between two components A and B, but in fact it is mediated by the Android system as shown in Fig. 3. The *Email* application in our running example communicates with a different application to display a web page in response to the user click on a hyperlink received in the email opened by an activity of the *Email* application. Let assume, there exists application *Browser* that can better serve *Email*, but the latter does not know if the former exists or can perform the desired action. For this operation, the activity A of *Email* starts *implicitly* the activity B of *Browser* without mentioning the name of target as the following: (1) activity A creates the intent (in Listing 1) with action `ACTION_VIEW` it wants to be performed and passes it by calling the method `startActivity()`. (2) The Android system receives the intent sent by A and looks for a relevant component among components of applications currently installed on the device. Let assume it matches (the filter in Listing 2 of) activity B in application *Browser*. (3) Android system activates B by calling its method `onCreate` with the intent received from A and as result, the *Browser* application displays the requested web page in a tab. This sequence of events appears to the user from a single application (*Email*) but in-fact it is rendered by two different applications namely *Email* and *Browser*. This all happens seamlessly to the user using ICC.

Even though, ICC is the major mechanism for inter-application interactions and contributes to rich user experience of mobile computing, but there are design limitations [1] in it and is one of the favourite targets to security threats [7, 40]. In the next section,

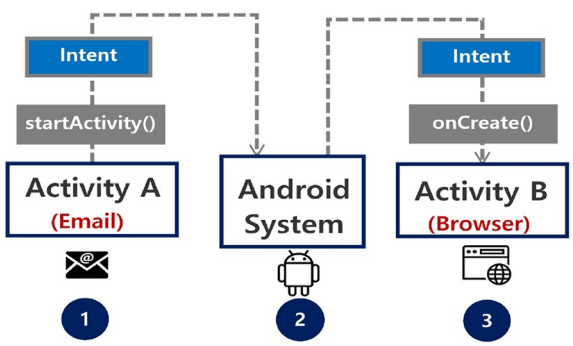

Fig. 3 Implicit inter-component communication [17]

formal definitions of intents, intent filters and inter-component communication are described and used in the formal reasoning about inter-component communication.

Formalizing inter-component communication

In the context of Android's ICC, when an application component intends to communicate with other component(s), it sends an explicit intent by providing the name of the component (if known), otherwise, an implicit intent (without the target name) is sent. Explicit intent does not require any evaluation and are simple to understand, while implicit intent requires a number of tests done by the system before an intent is passed on to an appropriate component. The existing specifications of intents, intent filters and the mechanism used to match a particular message request by a source component to a target component is informal and can not be used within formal proofs. In this section, the structure of implicit intent, intent filter and the mechanism to find the most appropriate component to receive implicit intent are formalized in theorem prover Coq. Such a formal definition can be used to prove theorems about intents, intent filters and inter-component communication using the Coq proof facility and the proof script can be checked mechanically using the Coq proof checker. The Coq source code and proofs of theorems are available on-line at [24].

Syntax of intent and intent filter

The main components of Android ICC are the data structures intent, intent filter and matching functions. An intent represents the content and the requested operation while the filter models a component. The data structures intent and intent filters are formalized in this sub-section and the matching functions are formalized in the next sub-section.

An implicit intent is a simple data structure containing the address and type of the data to act upon, the action to perform on data and some additional information. An intent is inductively defined in Coq as shown in Listing 3. The inductive definition consists of only one constructor (`int`, line 2) to construct elements (intents) of type `intent`. The constructor take as arguments an action, list of categories, URI and MIME type of the data. All the names (such as actions, categories, schemes, hosts and MIME types) are represented using the type `atom` defined in the library `Atom`, edited from [5]. The third argument of `intent` of type `uri` represents the content URI and is defined in Listing 4. A URI include optional elements *scheme*, *host*, *port* and *path*, represented by the arguments on lines 3, 4, 5 and 6, respectively. All the URI elements have type `atom` except *port* (line 5) which is modelled as a natural.

```
1 | Inductive intent:Type :=
2 |   int: atom → list atom → uri → atom → intent.
```

Listing 3 Data type intent

```
1 Inductive uri:Type :=
2  | url:
3   option atom →
4   option atom →
5   option nat →
6   option atom → uri.
```

Listing 4 Data type URI

To map an intent to a component, the elements of an intent are matched against the elements in the intent filter of a component defined in the *manifest* file of Android application. Intent filter `filter` is inductively defined in Listing 5. The only constructor `filt` of type `filter` gets four arguments: list of actions, list of categories, list of content URIs and MIME types (lines 3, 4, 5 and 6, respectively). The definitions of intent and intent filters are used to define Android application (line 1, Listing 6) at a high level modelled as a list of intents that it may invoke to integrate with other applications on the device. The mobile device environment (line 2, Listing 6) is the list of applications on the mobile device, represented by the list of filters in all the applications installed.

```
1 Inductive filter:Type :=
2   filt:
3   list atom →
4   list atom →
5   list uri →
6   list atom → filter.
```

Listing 5 Data type intent filter

```
1 Definition application := list intent.
2 Definition environment := list filter.
```

Listing 6 Android application and mobile device environment

Encoding Android application and environment

To demonstrate the applicability of our formal developments, a simple Android application and mobile device environment is created using formal notations defined. For simplicity, we assume the application consists of a single intent and the device contains a single application with just one intent filter. To realize this scenario, the intent and intent filter from Listings 1 and 2 are encoded in the formal notations developed in "Syntax of intent and intent filter" section. The encoding is used in a formal proof in next section.

```
1 Parameter ACTION_VIEW: atom.
2 Parameter CATEGORY_DEFAULT: atom.
3 Parameter CATEGORY_BROWSABLE: atom.
4 Parameter text_html: atom.
5 Parameter https: atom.
6 Parameter www_example_com: atom.
7 Parameter intent_html: atom.
8 Definition URL: uri := url (Some https) (Some www_example_com)
9  (Some 200) (Some intent_html).
```

Listing 7 Intent parameters

The intent ingredients including the action `ACTION_VIEW`, two categories `CATEGORY_DEFAULT` and `CATEGORY_BROWSABLE`, data type `text_html` and elements (scheme, host, port and path) of data URL are defined of type `atom` in Listing 7 (lines 5–7). The URL of the web page (line 1, Listing 1) is defined on lines 8–9 with

scheme, host, port number and path. All these parameters just defined are used to define intent and filter from Listings 1 and 2, respectively. In other words, the Coq definitions `exampleintent` and `examplefilter` of intent and filter (Listing 8) are the formal representations of the corresponding Java definitions in Listings 1 and 2. The implicit intent `exampleintent` is defined with action `ACTION_VIEW`, list of categories including `CATEGORY_DEFAULT` and `CATEGORY_BROWSABLE`, data URL and MIME type `text_html`. Similarly, the filter (for the *Browser* activity) is defined with a list of actions that it can perform, data categories, data URLs and types.

```
Definition exampleintent: intent :=
 int ACTION_VIEW (CATEGORY_DEFAULT::CATEGORY_BROWSABLE::nil)
   URL text_html.
Definition examplefilter: filter := filt (ACTION_VIEW::nil)
(CATEGORY_DEFAULT::CATEGORY_BROWSABLE::nil)
(URL::nil) (text_html::nil).
Definition exampleapp := exampleintent::nil.
Definition exampleenv := examplefilter::nil.
```

Listing 8 Encoding example application and environment

Finally, the example intent is used to define the application `exampleapp` (representing *Email*) and the filter is used to define the device environment `exampleenv`. The `exampleenv` represents the only application *Browser* on the device with one filter `examplefilter`. The major advantage of such formal definitions is that they can be used to mathematically reason about Android applications as demonstrated in "Proofs" section.

Intent resolution

After Android system receives an intent, it starts the appropriate target component for the intent based on the result of three tests [24] for action, category and data, respectively, against the corresponding elements in the intent filter of the target component. For an intent to resolve to a component, it must pass all these three tests. If the intent can not be resolved to any component, the source application may crash [17]. Following are the formal definitions of these three tests.

The action test is simple: the action in intent is matched against actions in the filter. This is modelled by the function `testaction` defined using the keyword `Definition` in Listing 9. The function definition in turn is using another function `find` which searches the action of type `atom` in intent (first argument) in the list of actions in filter (second argument). The space holder _ is used to represent arguments whose values are not important in the function body. The category test compares all the categories in the intent with the categories in the filter. The recursive function `testcategory` (Listing 10) takes a list of categories (of type `atom`) and categories listed in the intent filter and checks if all the categories in intent exist in the filter.

```
1 Definition testaction (action:atom) (f:filter):bool :=
2  match f with
3  | filt actions _ _ _ ⇒ find action actions
4  end.
```

Listing 9 Matching action

The third test, defined as a recursive function `testdata` in Listing 11, checks if the URI and MIME type of the content in the intent exist in the list of URIs and MIME types in the intent filter. Based on whether or not the URI and/or MIME type exists in the intent and/or filter, there are five different results. The first four cases in the function body (lines 4–10) correspond to the four rules on Android developers' website [19]. These (informal) rules are (formally) implemented using pattern matching on the four arguments namely, intent URI (`iuri`), intent type (`itype`), list of URIs in filter (`filuris`) and list of MIME types in filter (`filtypes`).

```
1  Fixpoint testcategory (intentcats:list atom)
2    (filtercats:list atom ) {struct intentcats}:bool :=
3    match intentcats with
4    | nil ⇒ true
5    | cons x l ⇒
6      match (find x intentcats) with
7      | false ⇒ false
8      | true ⇒ testcategory l filtercats
9      end
10   end.
```

Listing 10 Matching categories

```
1  Fixpoint testdata (iuri:option uri) (itype:option atom)
2   (filuris:list uri) (filtypes:list atom):bool :=
3   match iuri, itype, filuris, filtypes with
4   | None, None, nil, nil ⇒ true
5   | Some u, None, cons u' ul, nil ⇒
6      orb (testuri u' u) (testdata iuri None ul nil)
7   | None, Some it, nil, ftl ⇒ testtype ftl it
8   | Some u, Some it, cons u' ul, ftl ⇒
9      (orb (testuri u' u) (testdata iuri None ul nil)) &
10     (testtype ftl it)
11  | _, _, _, _ ⇒ false
```

Listing 11 Matching data

The first rule states that an intent with no URI and no MIME type passes the test if there are no URI and MIME types listed in the filter. This is represented on the line 4 where values of all the four arguments are `None` (`option` type) or `nil` (`list`). The second case (line 5) models rule **b** in the documentation, which states that an intent with a URI but no MIME type can be accepted only if its URI matches a URI pattern in the filter and the MIME type specification list is empty. The rule **c** is represented by the case 3 (line 7). An intent with only MIME type can pass the test if the same type exists in the list of types in the filter and there is no URI specification in the filter. The fourth rule is modelled by the case 4 (lines 8–10). An intent with URI and MIME type is accepted only if both, the URI and MIME type, matches with URI and MIME type in the filter. In

all other cases, such as `| None, None, cons u' ul, cons t' tl ⇒ false`, implicitly included in the code using space holders on line 11, the test fails.

The function `testdata` is calling two other functions `testtype` and `testuri` (lines 6, 7 and 9, Listing 11). For the first rule of function `testdata`, there is no URI and no MIME type and the result is `true`. For the next three rules, however, at least one of the URI and MIME type exists and the corresponding test function(s) is/are called. The test function `testtype` (Listing 12) is simple: it searches intent type in the list of filter types. If there is no intent type and likewise the filter does not require one (list of types in filter is empty), the test is passed. The formalization include explicit types and does not model implicit types. In the later case, the type would be inferred from the URI.

```
1 Definition testtype (filtypes:list atom)
2   (itype:atom):bool :=
3  match filtypes, itype with
4  | nil, _ ⇒ true
5  | filtypes, it ⇒ find it filtypes
6  end.
```

Listing 12 Matching type

```
1  Fixpoint testuri (filuri iuri:uri):bool :=
2   match filuri, iuri with
3   | url None None None None, _ ⇒ true
4   | url None _ (Some port) (Some path),
5      url _ _ porto patho ⇒
6      beq_nato (Some port) porto &
7      cmpoattr (Some path) patho
8   | url (Some scheme) None _ (Some path),
9      url schemeo _ _ patho ⇒
10     cmpoattr (Some scheme) schemeo &
11     cmpoattr (Some path) patho
12  | url None None (Some port) _,
13     url _ _ porto _ ⇒
14     beq_nato (Some port) porto
15  | url (Some scheme) None None None,
16     url (Some scheme') _ _ _ ⇒
17     scheme =?= scheme'
18  | url (Some scheme) (Some host) _ None,
19     url (Some scheme') (Some host') _ _ ⇒
20     (scheme =?= scheme') & (host =?= host')
21  | url (Some scheme) (Some host) _ (Some path), url
22     (Some scheme') (Some host') _ (Some path') ⇒
23     (scheme =?= scheme') &
24     (host =?= host') &
25     (path =?= path')
26  | url schemeo hosto porto patho,
27     url schemeo' hosto' porto' patho' ⇒
28     cmpoattr schemeo schemeo' &
29     cmpoattr hosto hosto' &
30     beq_nato porto porto' &
31     cmpoattr patho patho'
32  end.
```

Listing 13 Matching URI

The definition of `testuri` is shown in Listing 13. It gets the list of filter URIs and the intent URI as arguments and compares the intent URI to a URI specification in the filter by comparing it only to the parts of URI included in the filter. In the first case,

no URI in the intent, test is passed. The next six cases (lines 4–25) models the text *If a scheme is not specified . . . authority, and path pass the filter.* in the official documentation [19]. The last case (lines 26–31) represents the tests for all other cases: the axillary function `cmpoattr` compares the optional filter URI attribute with an intent URI attribute. For the attribute test to pass, the intent must include the same attribute listed in the filter and for the URI test to pass, all the attributes must pass the attribute test.

```
1 Definition resolve (i:intent) (f:filter):bool :=
2  match i, f with
3  | int a cl u t, filt fal fcl ful ftl ⇒
4     testaction a (filt fal fcl ful ftl) &
5     testcategory cl fcl &
6     testdata (Some u) (Some t) ful ftl
7  end.
```

Listing 14 Intent resolution

Finally, all the tests are combined together in function `resolve` defined in Listing 14. This function gets an intent and a filter and returns true if all the tests, namely `testaction`, `testcategory` and `testdata`, are passed. In other words, given an intent and a filter, the formal developments enable one to check if an intent would resolve to (accepted by) a component.

Application crash-safety

After intent, intent filter and intent resolution are defined, Android applications' safety against crashes due to intents is formally defined. The definition `resolve` is used to formally describe crash-safety property `intent_crash_safety` of an intent (Listing 15). This property is too strong: it defines an intent is crash safe if it must be accepted by the filter. To define crash-safety of an intent with respect to the entire device, a recursive function `intent_crash_safetey_env` is defined in Listing 16. An intent is crash-safe in the device if there exists at least an application (filter) that accepts the intent.

```
1 Definition intent_crash_safety (i: intent) (f: filter) :=
2  resolve i f.
```

Listing 15 Crash-safe intent

```
1 Fixpoint intent_crash_safetey_env (i: intent) (e: environment) :
2   bool :=
3 match e with
4 | nil => false
5 | f::tl =>
6   orb (intent_crash_safety i f) (intent_crash_safetey_env i tl)
7 end.
```

Listing 16 Crash-safe intent wrt. to the device

```
Fixpoint crash_safe_app (a: application) (e: environment) : bool
      :=
match a with
| nil => true
| i::tl => andb (intent_crash_safetey_env i e) (crash_safe_app
    tl e)
end.
```

Listing 17 Crash safe application

Finally, a crash-safe application is defined in Listing 17. An application (represented by the intents it may invoke) is crash-safe if every intent it may invoke resolves to an application component (represented by a filter) on the device. In the next section ("4" section), we formally prove that the example Android application with invoked intent `exampleintent` does not crash in the context of another application `examplefilter` on the device.

Proofs

In the previous sections, Android intent and intent filter were formalized and a high level formal model of Android ICC was built. The formalized definitions are rigorous, precise and can be used to formally reason about the ICC using computer-aided verification tool Coq. Using the logic of Coq proof assistant, formal proofs can be carried and checked mechanically using Coq proof checker. To demonstrate the significance of formal developments carried in "Formalizing inter-component communication" section, following formal proofs are carried out in this section:

- Proof of crash-safety of a simple real-life (Email) Android application ("Proof of application crash-safety" section),
- Proof of broadcast intent delivery to every application component ("Proof of broadcast resolution" section), and
- Proof of robust extension of component filters ("Robust environment extension" section).

Proof of application crash-safety

When an application invokes an intent, there must be an application on the mobile device to handle the intent, otherwise, the application that invoked the intent will crash [20]. Android platform 'guarantees' an intent must resolve to an application. The existing enforcement mechanism is to test, there exists an activity to respond to the invoked intent, when the source application activity first starts. The Java code to perform such a test is listed in Listing 18. The value of the *boolean* variable *isIntentSafe* determines if invoking the intent is safe for the application. Such tests are performed dynamically at run time after the application is deployed which is too late to avoid bad user experience. Furthermore, there is no proof of guarantee of robustness of the test and hence no proof of guarantee of application crash safety. In this section, we formally prove for the *Email* application whether or not the intent it generates resolves to the application *Browser*. It follows the proof of crash-safety of the application built from the example intent (by the *Email* app) in the environment created from the example filter (by the *Browser* app).

```
PackageManager manager = getPackageManager();
List<ResolveInfo> activities =
manager.queryIntentActivities(intent,manager.MATCH_DEFAULT_ONLY);
boolean isIntentSafe = activities.size() > 0;
```

Listing 18 Intent resolution test in Java [20]

For the first proof, a general proof of intent resolution is carried (theorem `resolve_intent`, Listing 19) which is used in the proof of resolution of our example intent `exampleintent` to filter `examplefilter` from Listing 8 (theorem `exampleintent_res_to_examplefilter`, Listing 20). Theorem `resolve_intent` defined in Listing 19 states that given an intent and a filter, if the intent passes action, category and data tests, it will be accepted by the application component with the filter defined in its manifest file. Note that the intent is constructed using the intent fields (the arguments of constructor `int` in Listing 3). This theorem is proved by first unfolding the definition of function `resolve` (using tactic `unfold`) and then using rewriting and simplification by Coq commands (tactics) `rewrite` and `simpl`, respectively. The proof of this theorem begins with keyword `Proof` at line 6 and ends at line 13. The last tactic `Qed` adds the proof to internal database for later retrieval in other proofs (see proof in Listing 20 for an example).

```
1  Theorem resolve_intent: forall a cl u t fal fcl ful ftl,
2    testaction a (filt fal fcl ful ftl) = true →
3    testcategory cl fcl = true →
4    testdata (Some u) (Some t) ful ftl = true →
5    resolve (int a cl u t) (filt fal fcl ful ftl) = true.
6  Proof.
7   intros ???????? TA TC TD.
8   unfold resolve.
9   rewrite TA.
10  rewrite TC.
11  rewrite TD.
12  simpl.
13  auto.
14 Qed.
```

Listing 19 Theorem 1–intent resolution

```
1  Theorem exampleintent_res_to_examplefilter:
2    intent_crash_safety exampleintent examplefilter = true.
3  Proof.
4   unfold intent_crash_safety.
5   apply resolve_intent; simpl.
6   (*CASE-testaction*)
7     rewrite eq_dec_atom_same; simpl; auto.
8   (*CASE-testcategory*)
9     do 2 rewrite eq_dec_atom_same.
10    destruct eq_atom; simpl; auto.
11  (*CASE-testdata*)
12    do 4 rewrite eq_dec_atom_same; simpl; auto.
13 Qed.
```

Listing 20 Theorem 2–proof of example intent resolution

The proof of theorem `exampleintent_res_to_examplefilter` in Listing 20 formally verifies that the intent `exampleintent` resolves to filter `examplefilter`. This theorem is proved by first unfolding the definition of `intent_crash_safety` and then applying the theorem `resolve_intent`. This opens up three sub-goals

which are closed by case analysis on the definition of equality using tactic `destruct` and rewriting an axillary lemma `eq_dec_atom_same` (see source code [24] for all definitions and proofs).

Finally, the example application *Email* modelled as `exampleapp` from Listing 8 is proved crash-safe in the device environment `exampleenv` with one application *Browser* modelled as `examplefilter` in Listing 8. The theorem `exampleapp_is_crash_safe_app` (Listing 21) states that the *Email* application can safely invokes the intent for opening a URL in a browser tab by requesting the *Browser* application installed on the mobile device.

```
Theorem exampleapp_is_crash_safe_app:
  crash_safe_app exampleapp exampleenv = true.
Proof.
 unfold crash_safe_app.
 simpl.
 do 7 rewrite eq_dec_atom_same.
 simpl.
 destruct eq_atom; simpl; auto.
Qed.
```

Listing 21 Theorem 3–proof of crash-safety of example application

Proof of broadcast resolution

A broadcast is a message wrapped in an intent and is received by any application that has declared a broadcast receiver, most commonly, in its manifest file. The method `sendBroadcast(intent)` is used to send a broadcast intent to multiple receivers. Using the developed formal setting, a formal proof that a broadcast message is indeed received by the applications subscribed for it (by declaring a receiver), is carried in Coq theorem prover.

```
Fixpoint broadcast (i: intent) (e: environment) : bool :=
 match e with
| nil => true
| cons f tl => (resolve i f) & broadcast i tl
end.
```

Listing 22 Intent broadcast

```
Theorem broadcast_general: forall i f s,
 broadcast i s = true ->
 In f s ->
 resolve i f = true.
Proof.
 intros ?????.
 apply In_split in H0.
 destruct H0.
 destruct H0.
 rewrite H0 in H.
 rewrite -> broadcast_append in H.
 unfold andb in *.
 destruct (resolve i f).
  (*CASE true*)
  auto.
  (*CASE false*)
  inversion H.
Qed.
```

Listing 23 Theorem 4–broadcast intent resolution

To do this, first a broadcast message (intent) is precisely defined in Listing 22. A broadcast intent is accepted (resolved to) by every application inside Android device environment that has declared a receiver for it. For simplicity, we assume every application on the mobile device (system) has declared broadcast receiver. This concept is formalized using recursive function `broadcast` in Listing 22. It takes an intent and system environment (list of applications' filters) and returns *true* if the intent is accepted by every application on the device.

Theorem 4 (Listing 23) states if an intent in a system is broadcast, it must resolve to (accepted by) every application on the system. This theorem is proved using rewriting, unfolding the definitions of function `broadcast` and then using case analysis on the `boolean` argument of `andb`. This later would generate two sub-goals: the first one is easy and is closed using tactic `auto` and the second one is closed using `inversion`. In addition to tactics used in earlier proofs, this proof is using an axillary lemma `broadcast_append`, tactics `In_split` and `inversion`.

Robust environment extension

Given a mobile environment and an application, it was formally verified in "Proof of application crash-safety" section that the application is safe in the environment. With the passage of time, it is normal for the user to install more applications and/or extend the intent filters of existing ones. It is necessary to ensure such an extension does not invalidate the guarantee previously provided. To further highlight the usefulness of our formal framework, we carry formal proof of application safety (Listing 24) and then prove that extending the filter with more categories is safe (Listing 25).

```
Definition aURL (s h p: atom) : uri :=
  url (Some s) (Some h) (Some 200) (Some p).

Definition anintent (a c p s h: atom) : intent :=
  int a (c::nil) (aURL s h p) p.

Definition afilter (a c p s h: atom) : filter :=
 filt (a::nil) (c::nil) (aURL s h p::nil) (p::nil).

Theorem aninttent_resoves_to_filter: forall a c p s h,
  resolve (anintent a c p s h) (afilter a c p s h) = true.
Proof.
 intros. simpl.
 do 5 rewrite eq_dec_atom_same.
 destruct eq_atom; simpl; auto.
Qed.
```

Listing 24 Intent resolution to a filter

```
1  Definition afilterplus (a c p s h: atom) (cats: list atom):
       filter :=
2   filt (a::nil) (c::cats) (aURL s h p::nil) (p::nil).
3
4  Theorem aninttent_resoves_to_filterplus: forall a c p s h cats,
5    resolve (anintent a c p s h) (afilterplus a c p s h cats) =
         true.
6  Proof.
7   intros. simpl.
8   do 5 rewrite eq_dec_atom_same.
9   destruct eq_atom; simpl; auto.
10 Qed.
```

Listing 25 Intent resolution to an extended filter

The Coq script in Listing 24 defines a general URL, intent and a filter. The theorem `aninttent_resoves_to_filter` formally proves that the intent resolves to (be accepted) by the components with filter `afilter`. In Listing 25, it is checked that the same intent also resolves to an extended (with categories `cats`) intent filter `afilterplus`. In other words, the proof of theorem `aninttent_resoves_to_filterplus` in Listing 25 confirms that if further categories are added to a filter, the application sending the intent is still safe and will not crash. Note that, variables `a, c, p, s, h, cats` used in definitions in Listings 24, 25 range over any action, category, path, scheme, host and categories, it generalizes the scope of the formal guarantees to any application and system.

Evaluation of the formal model

Demonstrating the usability of formal definitions of intent, filter and ICC, the proofs given in Listings 20, 21, 23, 24 and 25 statically (before executing them) guarantees that the applications invoking the intent will never fail given the conditions. These conditions are formally stated on lines 2–3 for the last theorem in Listing 23. For the theorems in Listings 20 and 21, there is no condition (hypothesis) and the proofs are guaranteed for the specific example intent and filter. The major advantage of proofs carried in mechanical theorem prover Coq is that the proof scripts are rigorous and their correctness can be checked using computer.

Formal modelling and proofs using interactive theorem prover, such as Coq, are more expressive and powerful than conventional simulation and formal proofs using model checking [10]. Simulation and model checking based approaches can be used to prove functional properties, however, they cannot be used to guarantee safety and security properties or can prove but with limited scope. Formal methods based on interactive theorem proving overcomes the disadvantages (e.g., state explosion problem) of model checking [3]. As the formal model CrashSafe has been defined in interactive theorem prover Coq, the formal developments and proofs are rigorous, reliable, have wide spread scope and can be mechanically checked. It can be used to prove safety properties of Android applications and can serve as a formal specification of Android ICC. The formal model, for example, can be applied to mathematically prove the shift of JavaScript process from a browser application on to a cloud-based computer for better performance [25] preserves the intended functionality.

Related work

There is a body of research carried to support secure mobile computing [34] in general and Android systems security [13, 14], in particular. In this section, we highlight the most relevant research related to ICC that is focused on application failures and security issues involving intents. From Android's mobile platform's perspective, ICC using intents is the major message passing technique among Android components and one of the major sources of application failures: both data and control can flow between the components, which can be exploited to leak secret information such as contacts. Android security in terms of data and control flows through ICC have been studied using static [9, 32, 35, 40] and dynamic [6, 11] analysis techniques.

Application crashes due to ICC

The most relevant research work is the study of Android applications crashes due to ICC. Ye et al. built an automated testing tool DroidFuzzer [41] to find bugs in Android applications. The tool, tailored towards activity components of applications, creates (crash) logs of abnormal data based on the data types (video and audio) a target component can accept as described in its intent filter. The tool was evaluated against three applications, two music players and one browser, using Android emulator and found 14 bugs. DroidFuzzer is limited only to activity component of Android applications and addresses applications crashes caused by the type of data. The formal analysis in this paper, on the other hand, studies crashes in any type of application component caused by any parameter of the intent (including data type, URL, action and category). A similar tool, intent fuzzer [35], was built for fuzzing inter-component communication in Android. Similar to DroidFuzzer, intent fuzzer generates a set of empty intents that a component can receive based on action and data type parameters. Inter fuzzer is rich in terms of coverage as it considers action, in addition to data type considered in DroidFuzzer. The most recently developed tool is CrashScope [30]. CrashScope automatically test Android applications using a systematic input generation based on several static and dynamic strategies and generates detailed crash reports in natural language formate. Another tool VanarSena [33] was developed for reporting crashes in applications by dynamically analysing applications' behaviour at runtime. The tool has been extensively tested against 3000 applications and found 2969 bugs. As VanarSena has been developed for and tested against Windows applications, we consider it orthogonal to our work.

The first fundamental difference between these tools and the work in this paper is that the former find bugs in applications using their data receiving capabilities (by generating several input pattens) while our work checks if an application crashes based on the receiving capabilities of other applications. The second difference is that these tools creates set of intents from the given data using mutation while our work generalizes on *all* possible intents using universal quantification. The third, and the most important, difference is that the approaches described do not have formal foundation and hence can be used to detect presence of failures/crashes but the formal model built in Coq can be used to prove absence of crashes in applications, the later being more power full approach.

Tools and frameworks for ICC analysis

Amandroid [40] is a framework that is tailored for Android applications to analyse their inter-component data flow. It captures both control and data flows by building a precise flow and context-sensitive inter-component data flow graph for application. The graph includes ICC edges and data and control can flow through these edges. A flow and context-sensitive algorithm is used to match the inter-component call source to the target. The flows are tracked by first inferring parameters for Android API calls for ICC, then resolve to the target (implicit or explicit) component(s) and track the flow from the source to target. There are other tools such as JarJarBinks [28] used to find bugs and study robustness of ICC in Android applications.

Chin et al. provided a tool ComDroid [9] for Android applications analysis. They analysed Davlik bytecode and detected vulnerabilities in inter-application communication of Android. Based on static analysis, ComDroid can detect a range of vulnerabilities including intent spoofing, broadcast theft, activity and service hijacking. Bugliesi et al. [7] developed a formal framework for the analysis of ICC. Their framework is based on typing techniques and include a formal calculus to reason about ICC. They implemented a prototype called Lintent that performs security type checking for Android applications.

Methods and techniques for ICC analysis

Later on, inspired from ComDroid [9], Octeau et al. [31] adopted a more general approach and developed a solver using a declarative language COAL. Using COAL, the inter-component objects are modelled and the required values of objects are inferred by taking the correlation between object fields. Even though, the solver applied to Android, it can be used in general static program analysis, in particular, where values of objects need to be inferred. Li et al. proposed a static taint analyser IccTA [26] for detecting leaks in ICC by tainting data. IccTA work on Dalvik bytecode and can detect inter-component based privacy leaks by providing a control-flow graph.

Addressing ICC as an instance of interprocedural distributive environment, Octeau et al. [32] developed a static analysis technique. In their approach, a specification is identified for every source and target of the ICC. The specification include values such as component name, action, category and data type and infers the missing values. By using [32], an analysis tool Epicc has been developed and more than a thousand applications were analysed for ICC vulnerabilities. Epicc analyses retargeted Java bytecode and does not handle URIs and content providers.

The state-of-the-art research work addresses Android security in terms of information leaks [26, 40] through ICC or finds bugs and vulnerabilities in ICC [7, 9, 28, 32]. On the contrary, CrashSafe checks application's safety against crashes caused by ICC. The tools DroidFuzzer [41], intent fuzzer [35], CrashScope [30] and VanarSena [33] investigate Applications' safety against crashes caused by ICC, however, their results can not be formally guaranteed. Furthermore, they are limited only to activities of applications [41], lack support for Android applications [33], or addresses applications' failures based on their own receiving capabilities. CrashSafe, on the other hand, has a formal foundation which makes it more rigorous, powerful and allows one to mathematically reason about all components of applications.

Conclusion and future research

Inter-component communication is the major mechanism for Android applications to share data and services with each other through messages called intents. An Android application crashes if it invokes an intent that cannot be received by any application on the phone. Application failures frustrates users and push them towards competitors, therefore, developers need to test their applications before deployment to avoid unpredictable behaviour and crashes at runtime. The official documentation for inter-component communication is not rigorous and hence can not be precisely studied, interpreted and used to reason about inter-component communication. In this paper, a formal model of the inter-component communication, dubbed as CrashSafe, was built in theorem prover Coq. The formal developments include formal definition of intents, applications (modelled as intent filters) and definitions of conditions to receive messages. The formal notations were used to encode simplified versions of an *Email* and a *Browser* application and it was proved the *Email* application can safely request the *Browser* application to open a web page in a tab. The formal model, in addition, was used in proving the correctness of the intent broadcasts. The formal model CrashSafe enables one to (i) check the correctness of inter-component communication in Android systems and (ii) establishes a formal foundation for other tools to assess Android applications' reliability and safety, while at the same time it is simple to understand and use as formal specification. The widespread use of Android in the global market makes our formal developments useful for most of the mobile device users.

Dimensions of future research The formal proof of crash-safety was carried for Android applications represented at a high level. It would be more interesting had the proof been carried for applications including all the elements (e.g., all the filters and intents involved in the *Email* and *Browser* applications). The current model is targeted towards Android system and hence cannot be used to asses applications' reliability developed using other popular platforms such as Apple's iOS. To assess the formal model and increase its practicality in real-life environment, a proof-of-concept tool implementing the logic behind the formal model needs to be developed.

Abbreviations

iOS: Apple's operating system; SDK: software development kit; API: application programming interface; ICC: inter-component communication; TV: television; APK: application package; ID: identifier; URI: uniform resource identifier; MIME: multipurpose internet mail extensions; URL: universal resource locator; COAL: computer organization and assembly language.

Authors' contributions

WK, AA and HU defined the Coq formalization, carried Coq proofs and wrote the paper in LaTeX. KS, AJA and SDK investigated the literature and created Figs. 1 and 3. The definitions of *Email* and *Browser* applications in Java were defined and encoded in the formal syntax by MA and SA. All the authors reviewed the existing paper and helped improving its structure. All authors read and approved the final manuscript.

Author details

[1] COMSATS University, Islamabad, Wah Campus, Wah Cantt, Pakistan. [2] College of Computer Science and Engineering, University of Hail, Hail, Kingdom of Saudi Arabia.

Acknowledgements

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

We have provided Coq source code and proofs of theorems—as extended details for interested readers—in [24].

Funding
Not application.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 23 April 2018 Accepted: 6 July 2018
Published online: 26 July 2018

References

1. Ahmad W, Kästner C, Sunshine J, Aldrich J (2016) Inter-app communication in android: developer challenges. In: Proceedings of the 13th international conference on mining software repositories. ACM, New York, pp 177–188
2. Apple Inc (2017) Apple Apps Store. https://itunes.apple.com/us/genre/ios/id36?mt=8. Accessed June 2017
3. Armstrong RC, Punnoose RJ, Wong MH, Mayo JR (2014) Survey of existing tools for formal verification. Tech Rep, Sandia National Laboratories
4. Arshad H, Jantan AB, Abiodun OI (2018) Digital forensics: review of issues in scientific validation of digital evidence. J Inf Process Syst 14(2):346–376
5. Aydemir B, Charguéraud A, Pierce BC, Pollack R, Weirich S (2008) Engineering formal metatheory. In: Acm sigplan notices, vol 43. ACM, New York, pp 3–15
6. Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi AR, Shastry B (2012) Towards taming privilege-escalation attacks on android. In: NDSS, vol 17, p 19
7. Bugliesi M, Calzavara S, Spanò A (2013) Lintent: towards security type-checking of android applications. In: Formal techniques for distributed systems. Springer, Berlin, pp 289–304
8. Chaudhuri A (2009) Language-based security on android. In: Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security. ACM, New York, pp 1–7
9. Chin E, Felt AP, Greenwood K, Wagner D (2011) Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. ACM, New York, pp 239–252
10. Clarke EM, Grumberg O, Peled D (1999) Model checking. MIT press, Cambridge
11. Dietz M, Shekhar S, Pisetsky Y, Shu A, Wallach DS (2011) Quire: lightweight provenance for smart phone operating systems. In: USENIX security symposium, vol 31
12. Dinh HT, Lee C, Niyato D, Wang P (2013) A survey of mobile cloud computing: architecture, applications, and approaches. Wireless Commun Mobile Comput 13(18):1587–1611
13. Enck W, Ongtang M, McDaniel P (2009) Understanding android security. IEEE Secur Privacy 7(1):50–57
14. Faruki P, Bharmal A, Laxmi V, Ganmoor V, Gaur MS, Conti M, Rajarajan M (2015) Android security: a survey of issues, malware penetration, and defenses. IEEE Commun Surv Tutorials 17(2):998–1022
15. Felt AP, Chin E, Hanna S, Song D, Wagner D (2011) Android permissions demystified. In: Proceedings of the 18th ACM conference on computer and communications security. ACM, New York, pp 627–638
16. Google Inc. Android Apps on Google Play. https://play.google.com/store/apps?hl=en
17. Google Inc. Android. https://www.android.com/
18. Google Inc (2017) Application fundamentals. https://developer.android.com/guide/components/fundamentals.html. Accessed June 2017
19. Google Inc (2017) Intents and intent filters. https://developer.android.com/guide/components/intents-filters.html. Accessed June 2017
20. Google Inc (2018) Sending the User to Another App. https://developer.android.com/training/basics/intents/sending.html. Accessed May 2018
21. Hall SP, Anderson E (2009) Operating systems for mobile computing. J Comput Sci Coll 25(2):64–71
22. INRIA. The Coq Proof Assistant. https://coq.inria.fr/
23. Kantola D, Chin E, He W, Wagner D (2012) Reducing attack surfaces for intra-application communication in android. In: Proceedings of the second ACM workshop on security and privacy in smartphones and mobile devices. ACM, New York, pp 69–80
24. Khan W, Habib U, Akash A, Khalid S, Sultan Daud K (2017) Coq Script. https://github.com/wilstef/android-ipc. Accessed June 2017
25. Kim D (2017) Cloud computing to improve Javascript processing efficiency of mobile applications. J Inf Process Syst 13(4):4
26. Li L, Bartel A, Bissyandé TF, Klein J, Le Traon Y, Arzt S, Rasthofer S, Bodden E, Octeau D, McDaniel P (2015) Iccta: detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th international conference on software engineering, vol 1. IEEE Press, pp 280–291
27. Maheshwari S (2018) That game on your phone may be tracking what you're watching on tv. https://www.nytimes.com/2017/12/28/business/media/alphonso-app-tracking.html
28. Maji AK, Arshad FA, Bagchi S, Rellermeyer JS (2012) An empirical study of the robustness of inter-component communication in android. In: 2012 42nd annual IEEE/IFIP international conference on dependable systems and networks (DSN), pp 1–12
29. Meier S, Schmidt B, Cremers C, Basin D (2013) The tamarin prover for the symbolic analysis of security protocols. In: International conference on computer aided verification. Springer, Berlin, pp 696–701
30. Moran K, Linares-Vásquez M, Bernal-Cárdenas C, Vendome C, Poshyvanyk D (2017) Crashscope: a practical tool for automated testing of android applications. In: 2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C), pp 15–18

31. Octeau D, Luchaup D, Dering M, Jha S, McDaniel P (2015) Composite constant propagation: application to android inter-component communication analysis. In: Proceedings of the 37th international conference on software engineering, vol 1, IEEE Press, pp. 77–88
32. Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Le Traon Y (2013) Effective inter-component communication mapping in android with epicc: an essential step towards holistic security analysis. In: Proceedings of the 22nd USENIX security symposium, pp. 543–558
33. Ravindranath L, Nath S, Padhye J, Balakrishnan H (2014) Automatic and scalable fault detection for mobile applications. In: Proceedings of the 12th annual international conference on mobile systems, applications, and services. ACM, New York, pp 190–203
34. Sajjad M, Abbasi AA, Malik A, Altamimi AB, Alseadoon IM (2018) Classification and mapping of adaptive security for mobile computing. IEEE Trans Emerg Topics Comput. https://doi.org/10.1109/TETC.2018.2791459
35. Sasnauskas R, Regehr J (2014) Intent fuzzer: crafting intents of death. In: Proceedings of the 2014 joint international workshop on dynamic analysis (WODA) and software and system performance testing, debugging, and analytics (PERTEA). ACM, New York, pp 1–5
36. Satyanarayanan M (2010) Mobile computing: the next decade. In: Proceedings of the 1st ACM workshop on mobile cloud computing & services: social networks and beyond. ACM, New York, p 5
37. Schmidt AD, Schmidt HG, Clausen J, Yuksel KA, Kiraz O, Camtepe A, Albayrak S (2008) Enhancing security of linux-based android devices. In: Proceedings of 15th international Linux Kongress. Lehmann
38. Shao J, Kuk G, Terrell M, Chen S (2014) Why do applications request my contacts data? a large-scale study on openness and control of user contacts permission in android mobile applicaitons marketplace. Front Bus Res China 8(1):113–135
39. The Statistics Portal. Global mobile OS market share 2009–2017, by quarter. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/
40. Wei F, Roy S, Ou X et al (2014) Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. ACM, New York, pp 1329–1341
41. Ye H, Cheng S, Zhang L, Jiang F (2013) Droidfuzzer: fuzzing the android apps with intent-filter tag. In: Proceedings of international conference on advances in mobile computing & multimedia. ACM, New York, p 68

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com