IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 7, 2018, accepted January 11, 2019, date of publication January 25, 2019, date of current version February 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2895261

Formal Analysis of Language-Based Android Security Using Theorem Proving Approach

WILAYAT KHAN[1], MUHAMMAD KAMRAN[2], AAKASH AHMAD[3], FARRUKH ASLAM KHAN[4], (Senior Member, IEEE), AND ABDELOUAHID DERHAB[4]

[1]Department of Electrical and Computer Engineering, COMSATS University Islamabad at Wah Campus, Wah Cantonment 47040, Pakistan
[2]Department of Computer Science, COMSATS University Islamabad at Wah Campus, Wah Cantonment 47040, Pakistan
[3]College of Computer Science and Engineering, University of Hail, Ha'il 55476, Saudi Arabia
[4]Center of Excellence in Information Assurance, King Saud University, Riyadh 11653, Saudi Arabia

Corresponding author: Wilayat Khan (wilayat@ciitwah.edu.pk)

This work was supported by the Deanship of Scientific Research at King Saud University, Saudi Arabia, through the Research Group under Project RGP-214.

ABSTRACT Mobile devices are an indispensable part of modern-day lives to support portable computations and context-aware communication. Android applications within a mobile device share data to support application operations and better user experience, which also increases security risks to device's data integrity and confidentiality. To analyze the security provided by the Android permissions, modern security techniques, based on the programming languages, have been used to enforce best practices for developing the secure Android applications. Android security assessment, based on the language-based techniques in an informal setting without formal tool support, is tedious and error-prone. Furthermore, the lack of proof of the soundness of the language-based techniques raises questions about the validity of the analysis. To enable computer-aided formal verification in Android security domain, we have developed a mathematical model of language-based Android security using computer-based proof assistant Coq. One of the main challenges for mechanizing the language-based security in theorem prover relates to the complexity of variable bindings in language-based security techniques. As the main contributions of the paper: 1) the language-based security, including variable binding, is formalized in theorem prover Coq; 2) a formal type checker is built to type check (capture safe data flows within) Android applications using computer; and 3) the soundness of the language-based security technique (type system) is mechanically verified. The formal model of the Android type system and their proof of soundness are machine-readable, and their correctness can be checked in the computer using Coq proof and type checkers.

INDEX TERMS Android security, formal verification, language-based security, locally nameless representation, machine-readable proofs, theorem proving.

I. INTRODUCTION

Originally designed and built for remote conversation, modern mobile devices are now mini computers with support for third party applications' execution. This feature makes mobile devices *smart* and opens up the gate for applications supporting various tasks. These tasks include, but are not limited to, mobile commerce, health monitoring, entertainment, and location querying [1]. According to Statista [5], there are over two billions of mobile users around the globe and millions of mobile applications available on stores such as Google Play [2] and Apple Apps Store [3] for the most popular operating systems Android and iOS, respectively.

Android is an operating system and open source applications development platform developed by Google. Based on Linux kernel, Android is the most popular operating system running on mobile devices such as tablets and smart phones [4], [5]. According to Statista [5], Android market share in sales to end users in second quarter of 2017 was around 90%. Android framework is available with a software development kit (SDK) and many other developer tools. Its application programming interface (API) is used

The associate editor coordinating the review of this manuscript and approving it for publication was Feng Lin.

for building innovative modular mobile applications. The Android API allows applications to access software and hardware resources, such as such as camera and contacts, of the mobile device. Furthermore, a mobile application can share data with other applications or components within the same application using the API.

The data sharing facility available on Android framework provides enhanced user experience but it may also be exploited by the attackers. An unauthorised use of resources may lead to numerous security issues: a malicious application, for example, can illegally access contacts and share them with others by sending messages using Internet. The modular Android application framework enables mutually non-trusting Android applications, from different owners, to share their functionalities and resources. Security inside Android powered mobile devices is provided by the enforcement mechanism based on permissions [6], [7]. Android permissions control the flow of data (such as contacts and photos) and access to resources (such as camera and Internet) within mobile devices. Each application declares a set of permissions, that it would require to best serve the user, in a manifest file and are set by a package installer during application installation. The user grants (or denies) access to resources by giving (or refusing) the permission that the application requests based on the trust it has on application developer and the security requirements of the resource.

Even though a sophisticated security mechanism based on permissions is in place, it does not provide information flow guarantees [8]. The applications normally request permissions that can be misused to find the location of mobile (user) and put the mobile device and carrier network at risk [9], [10]. Moreover, the permission system can be circumvented by (1) spreading the permissions over different applications that communicate with each other [10], or (2) when the applications over *overprivileged* [6]. Even if the user denies a permission that is requested by an application, another application that holds that permission may perform the privileged task for the former application [11]. To study the effectiveness of Android's access control mechanism based on permissions, language-based technique has been applied to it [12], [13]. However, no proof of soundness of the language-based security technique is provided in a formal setting, which raises questions about the validity of the technique and tools [12] built on top of it.

In this paper, we build a formal model of Android permissions system and language-based security in mechanical theorem prover Coq, and carry out the correctness proof of the system in the theorem prover. The mechanized formal model is interesting from two perspectives: 1) it can be seen as an implementation of the security system (type checker) to mechanically verify (type check) safe data flows within simple Android applications, and 2) the mechanization of Android type system in Coq provides a foundation to reason about soundness properties. The later, and the most challenging due to variable binding, is demonstrated by proving two soundness properties of the type system.[1]

A. RESEARCH CONTEXT AND CHALLENGES

Android permissions system allows applications to interact and share data with each other; however, such data flows may result in unintended security consequences. Assume the data is labeled with security labels $\top$ (top), `READ`, `WRITE` and $\bot$ (bottom). When confidentiality of data is assessed, the security label $\top$ tags secret data that can be read by other components with label $\top$ only. The data with label $\bot$, on the other hand, is public with least security concerns: it can be read by anyone. When integrity is considered, the data with label $\top$ is the most trusted one and can be added to (or influence) data with any label. Similarly, the data with label $\bot$ is untrusted (tainted) and it must not be written to data with higher security concerns (such as `WRITE` and $\top$)

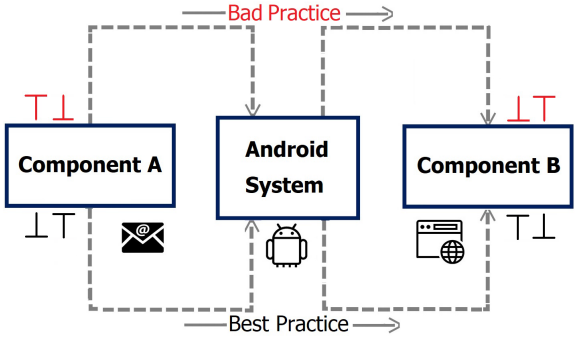

FIGURE 1. Data flow scenario: data flows from component `A` to `B`.

Consider a scenario, described in Fig. 1, where a component B reads data from the content provider (component) A. The security label pairs $\top\bot$ and $\bot\top$ (regardless of color) represent secret/trusted and public/tainted data, respectively, when data flow properties confidentiality/integrity are assessed. From component A's perspective, component `A` writes to/updates the data in content provider (component) `B`. The upper dashed arrows in the figure describes a security violation: the component `B` with security context defined at $\bot\top$ (right label pair in red), reads secret and write tainted data with label $\top\bot$, respectively, from the content provider `A` (left label pair in red). The security system should enforce the security policy [12]: *the component that reads from/writes to a content provider (component)* `A` *must have permission (label) at least as high as component* `A`. The data flow just described is a violation of the security policy (bad practice). The secret data is read by a component with security context $\bot$, which effectively can be read by any other component with label (permission) $\bot$ (secrecy violation). Similarly, the data in the content provider (component) `B` has integrity label $\top$ (trusted data), which is tainted by data that might have been written to (modified) by any component with label $\bot$

[1]All the formal definitions and proof scripts are available at Github repository at https://github.com/wilstef/secdroid.

(integrity violation). The bottom dashed arrows in Fig. 1 show an example of the best practice: public and trusted data can be read by a component with secret and untrusted security context (data flow from higher security context to lower is permitted). A number of other data flow scenarios in Android applications, which can be captured by the type system mechanized in this paper, have been described in [12].

To prevent such security violations, existing security enforcement systems [12], [13] can be used to type-check (capture valid data flows) and reason about Android applications. However, giving guarantees of data flow security properties of well-typed applications in an informal setting is tedious and error prone. Furthermore, these security techniques themselves need to be reasoned about to verify their correctness. As these security techniques cannot be read by a formal computer tool, they are not fit for mechanical reasoning about Android applications' security and the (programming) language-based technique itself. Among the major advantages of formal analysis using computer-aided formal verification tools is that a computer can be used to machine check correctness (well-typedness) of Android applications, analyze them mathematically and generate machine-readable proof script. Formally reasoning about programming language properties using mechanical theorem provers is challenging due to the issues with representation and manipulation of terms using variable binding [14]. To resolve such issues, locally nameless representation [15] has been applied to formalizing programming languages in theorem prover. Locally nameless representation has been applied to different domains in the past [16]–[18]. However, to the best of our knowledge, no one has used this approach in mobile applications' security domain.

B. SOLUTION OVERVIEW

In this paper, we build a formal model of language-based technique [13] in the human-assisted computer-based proof tool Coq using locally nameless representation. Furthermore, we demonstrate effectiveness of locally nameless representation in carrying out formal machine-readable proof of soundness of the language-based technique. The major contributions of this paper are the following:

- We enable formal reasoning about Android's permission system using computer-based theorem prover tool; a formal model of the Android's permissions is built using the logic behind theorem prover.
- We augment the language-based security technique with computer-aided verification tool, and locally nameless representation is formalized.
- We mechanically (using a computer) check correctness of terms (programs), and build a formal model of Android type checker.
- We demonstrate that locally nameless approach facilitates the proof process and guarantees well-typed (correct) Android code, and mechanically carry out formal proof of soundness of the Android type checker.

The rest of the paper is organized as the following: In the next section, a brief overview of the Android application security, permissions, Coq theorem prover and locally nameless representation is given. In Section III, the formal model of the type system is presented. The proofs of the type system properties, *weakening* and *strengthening*, are given in Section IV. A summary of the related work is presented in Section V, and the paper is concluded in Section VI.

II. BACKGROUND

In this section, Android application framework with the security model, the formal language describing Android applications, proof assistant Coq and locally nameless representation are briefly introduced. The formalization of Android permissions and language-based security technique are described in Section III

A. ANDROID SECURITY MODEL

The Android application framework [7] supports programming languages Kotlin, C++ and Java for application development. The source code of an Android application is compiled into an Android package (APK) using Android SDK tools. The package, containing all the contents of Android app, is used by the Android-powered device to install the app. Following Linux kernel security rules, Android operating system considers each app as a different user and assigns it a unique user ID. It executes each individual app in isolation by a separate process using its own virtual machine. The default rule for Android security is the principle of *least privilege*. By this rule, each app is given only the minimum permissions to access components required for its work. Different applications can also share data with each other (e.g., by sharing common Linux user ID) and access system resources by requesting permissions.

An Android application comprises of many components of four types: *activity*, *receiver*, *service* and *provider*. Each individual component provides a different entry point for the system to enter the application. The activity component represents the presentation layer of application. For example, the user interface on screen, which the user sees, is controlled by an activity. A single application may include more than one activity and a user can switch between activities. Services execute in the background and perform long-running operations without a user interface. However, they can notify the user through notifications. Broadcast receivers receive messages from system and receive implicit (without target name) intents. The intents are messages used by Android applications to request different functionalities from other services or activities. An application can address/access a component through intents *explicitly* by using the target component name or *implicitly* by naming an action to be performed by a relevant component. Content provider components serve as data storage units for the applications. Other application components can read/write the data from/to content provider upon permission.

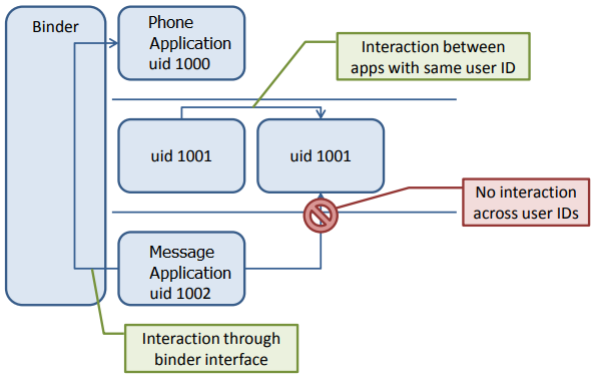

FIGURE 2. Android permission system [19].

Android enforces security through permissions by restricting applications to interact with each other and access different resources. For two applications to access each others' resources, they both must share the same user ID or they must have the desired permission to do so. Two applications, both with IDs 1001 in Fig. 2 (inside sandbox), has the same user ID (they are created by the same developer who has requested common ID) and can access each other's resources directly using the file system. All other applications, with different user ID, can interact with each other through the Android binder interface, which invokes permissions. To send an SMS, for example, the *Message* application (Fig. 2) requests the *Phone* application through the binder to send an SMS. The binder forwards the request only if the *Message* application has the permission *SEND_SMS*.

v	::=	$n \mid x \mid void$	values
i	::=	(n, v)	intent
t	::=		code
		$call(i)$	call activitiy
		$return(v)$	return from activity
		$register(\texttt{SEND}, \lambda x.t)$	register new receiver
		$send(\texttt{RECEIVE}, i)$	send to receiver
		$!n$	read from provider
		$n := v$	write to provider
		$let\ x = t\ in\ t'$	evaluate
		$\curlyvee t$	fork
		$t + t'$	choice
		v	result

FIGURE 3. Program syntax.

B. LANGUAGE-BASED ANDROID SECURITY

Chudhuri [13] defined a core programming language to describe Android applications. Assuming a security lattice $\sqsupseteq$ of permissions, the language syntax (Fig. 3) includes intents as a pair of component name (action) to be accessed and a value to be passed to the component as a parameter. All names n, variables x and a constant *void* make the values v. Terms t in the language are defined for call and return from an activity, bind to service, register new service, send to a receiver, reading from/writing to content provider and so on. A program runs in an environment, derived from the set of applications installed on the mobile device, and maps names to components. Avik Chudhuri defined a security type system to capture safe data flows among Android applications. For further details about the language syntax and type system, readers are recommended to refer to [13].

C. COQ PROOF ASSISTANT

To formally reason about systems, a formal model of the system and the property of interest is built in the logic of a theorem prover (such as Coq [20] and Isabelle/HOL [21]). The proof facility of the theorem prover is used to build a proof that the (model of the) system holds the property and the proof checker of the tool is used to mechanically check if the proof is valid. To describe formal developments and proofs using theorem prover, a simple system of numbers is defined and reasoned about using the tool Coq. To begin with, numbers are inductively defined as data type `nat` using the keyword `Inductive` with two constructors for generating elements of the type `nat` (lines 1–3, Fig. 4). The definition **nat** states that `O` (for 0) is `nat` and if `n` is `nat` then `S n` is also `nat`. The term `S (S (S O))`, for example, is a number (3) in `nat`.

```
1  Inductive nat : Type :=
2    | O : nat
3    | S : nat → nat.
4
5  Fixpoint add (n m: nat) : nat :=
6   match n with
7   | O ⇒ m
8   | S n′ ⇒ S (add n′ m)
9   end.
10
11 Lemma add_n_o: ∀ n, add n O = n.
12 Proof.
13    induction n.
14    (*CASE 1: n is O*)
15       reflexivity.
16    (*CASE 2: n is (S n)*)
17       simpl. rewrite IHn. auto.
18 Qed.
```

FIGURE 4. Interactive formal proof in Coq.

Next, we define a recursive function `add` (lines 5–9) on the numbers just defined. The function returns the second argument `m` if the first argument is `O` and it returns `S (add n′ m)` if the first argument is of the form `S n′`. A lemma `add_n_o`, that `add n O = n` holds for any value of `n`, is stated and proved in Fig. 4 (lines 11–18). The lemma is proved using induction on the construction of the first argument `n`. During the proof process, the theorem prover is guided interactively by providing commands called *tactics* (lines 13–17). The correctness of the proof script just created (lines 12–18 in Fig. 4) can be mechanically checked using the Coq proof checker program. The symbol → in the Coq script

is used to represent a function type (line 3) while the symbol ⇒ is used to return a value from a function (lines 7–8).

D. LOCALLY NAMELESS REPRESENTATION

For increased confidence on complex and large system designs in programming languages, it is often desired to check the proofs of properties of programming languages using human-assisted computer-based theorem provers, such as Coq and Isabelle/HOL. However, this comes with the key challenge of representation and manipulation of terms with variable bindings, in particular, in large formal developments. Aydemir *et al.* [14] introduced a novel approach to formalizing metatheory based on a combination of *locally nameless* [15] representation of terms involving binders and cofinite quantification of free variable names in rules involving binders in inductive definitions of relations on terms. In locally nameless representation, bound variables are represented with de Bruijn indices and free variables are represented with names. Locally nameless representation obviates the issues related to alpha-conversion as now each alpha-equivalence class of terms has a unique representation. Cofinite quantification, on the other hand, makes the structural induction principles of relations strong enough for metatheoretic reasoning [14], [15].

To show comprehensiveness of their approach, Aydemir *et al.* [14] created several large formal developments using Coq proof assistant. The mechanization of the formal language in this paper is built on top of the libraries developed by Aydemir *et al.* In the next section, formal definitions are introduced. For further details about the functions used, readers are advised to refer to proof assistant Coq libraries in [14].

III. FORMALIZING LANGUAGE-BASED ANDROID SECURITY

To protect against application-level attacks on modern computer systems, language-based security techniques are applied at programming language (application) level [22], [23]. The formal language for representing Android applications and the type system (language-based Android security) [13], enforcing best practices, cannot be read by computer-aided verification tool and hence are not fit for mechanical reasoning and computer-based checking. To enable computer-based checking and reasoning, the language and type checker both must be defined in the logic of a theorem prover. In this section, the formal language and the type system enforcing the best practices for Android applications development, as described in [13], are extended with locally nameless representation and formalized in the logic of theorem prover. The reason formal language and type system in [13] are chosen for mechanization is that they are simple and abstract while at the same time they are expressive enough to establish formal reasoning against Android permissions.

```
1 | Inductive Pm : Type :=
2 |     | CALL : Pm
3 |     | BIND : Pm
4 |     | READ : Pm
5 |     | WRITE : Pm
6 |     | SEND : Pm
7 |     | RECEIVE: Pm
8 |     | TOP : Pm
9 |     | BOTTOM : Pm.
```

FIGURE 5. Definition of Android permissions.

A. LANGUAGE SYNTAX

To begin with, Android permissions are defined as an inductive type `Pm` (Fig. 5) using Coq `Inductive` keyword of sort `Type`. These permissions (each make a constructor of type `Pm`) are the following: an application requires to have `CALL` permission in order to create an activity, `BIND` is needed for an application to bind to a service, `READ`/`WRITE` are required to read/write from/to a content provider, respectively. Similarly, an application can send data to a receiver only if it has `SEND` permission and the receiver likewise can receive it only if it has the permission `RECEIVE`. The two permissions `TOP` and `BOTTOM` bound other types above (highest authority) and below (lowest authority), respectively.

```
1  | Inductive ty: Type :=
2  |   | ty_stuck : ty
3  |   | ty_dnc : ty
4  |   | ty_any : Pm → Pm → ty
5  |   | ty_pro : Pm → Pm → ty
6  |   | ty_act : Pm → ty → ty → ty → ty
7  |   | ty_rvr : Pm → ty → Ty → ty with
8  |           Ty : Type :=
9  |         | Ty_bvar : nat → Ty
10 |         | Ty_fvar : atom → Ty
11 |         | Ty_sub : ty → Ty
12 |         | Ty_stack : ty → ty → Ty .
```

FIGURE 6. Security type system.

To track data flow within applications, a system of security types `ty` for the language is inductively defined (Fig. 6) to label data. The type `ty_stuck` is given to programs that are blocked due to access control and the type `ty_dnc` represents data whose security is of no concern.[2] The types `ty_pro`, `ty_act` and `ty_rvr` are, respectively, for components provider, activity and receiver and are given to names bound in the environment. Following [13], services are encoded with receivers; therefore, there is no separate type for services. The third and fourth arguments of type `ty` in `ty_act` represent stack type, which is given to the code run by an activity (window). Values returned to the current window have type `ty` (second argument).

Inside type `ty` is a mutually dependent type `Ty` with a subtype constructor `Ty_sub`, which would be used to

[2]This is, instead, represented by '_' in [13].

convert type `ty` to `Ty` (see below). The type system is extended with two constructors `Ty_bvar` and `Ty_fvar` for bound and free variables, respectively. The argument of type `atom` in `Ty_fvar` will be used to represent free names. The argument of type `nat` in `Ty_bvar` is used to model bound variables using de Bruijn indexes. In addition, it should be noted that the constructors in type `ty` in [13] are more specific while they are defined parametric in Fig. 6 to make them more general. Such a general representation enables to universally quantify over permission, type, number and name in the proofs, extending the scope of formal developments and proofs.

```
Inductive term : Set :=
  | bvar : nat → term
  | fvar : atom → term
  | lam : term → term
  | call : atom → atom → term
  | ret : atom → term
  | bind : atom → atom → term → term
  | reg : Pm → term → term
  | send : Pm → atom → atom → term
  | read : atom → term
  | write: atom → atom → term
  | lett : term → term → term
  | fork : term → term
  | choice : term → term → term
  | result : atom → term
  | void : term.
```

FIGURE 7. Definition of terms.

To write Android applications (programs), terms are inductively defined as shown in Fig. 7. To formally implement the locally nameless approach in Coq, bound and free variables in terms need to be distinguished. As mentioned before, de Bruijn indices and names are used to represent bound and free variables, respectively. The term `bvar` takes a natural number as the index to represent bound variables and `fvar` takes a name (of type `atom`) to represent free variables. The natural number index is the number of abstractions needed to be traversed to reach abstraction binding that variable. Lambda abstraction is formalized with constructor `lam` with the only argument defines the body of the abstraction. Calling an activity is modeled using `call`, which gets an activity name and a value that is passed to the activity. Unlike in [13] where name and value pair is separately defined as an *intent*, these parameters of type `atom` make the two arguments of constructor `call`. The type `atom` abstractly model both implicit and explicit intents where the identifier of type `atom` represents component name (to be accessed) or action (to be performed), respectively. Our formal setting captures data flow through intents either if the component is accessed by name or action. In other words, our type system captures flows through both implicit and explicit intents.

The program `ret` returns control by popping the current window off the stack and returns control to the previous activity. To bind to a service, the program `bind` is used, where the first two arguments make the intent and the third argument is the code to be executed. The program `reg` registers a new

```
Fixpoint fv (t : term) {struct t} : atoms :=
  match t with
    | bvar i ⇒ {}
    | fvar x ⇒ singleton x
    | lam t' ⇒ fv t'
    | call x1 x2 ⇒ singleton x1 ‘union‘ singleton x2
    | ret x ⇒ singleton x
    | bind x1 x2 t' ⇒ singleton x1 ‘union‘ singleton x2 ‘union‘ (fv t')
    | reg p t1 ⇒ fv t1
    | send p x1 x2 ⇒ singleton x1 ‘union‘ singleton x2
    | read x ⇒ singleton x
    | write x1 x2 ⇒ singleton x1 ‘union‘ singleton x2
    | lett t1 t2 ⇒ (fv t1) ‘union‘ (fv t2)
    | fork t' ⇒ fv t'
    | choice t1 t2 ⇒ (fv t1) ‘union‘ (fv t2)
    | result x ⇒ singleton x
    | void ⇒ {}
  end.
```

FIGURE 8. Definition of function for collecting free variables.

receiver with a body (second argument) and sets it with a permission (first argument). To send intent to a receiver, the program `send` is used where the first argument of type `Pm` is the permission required and last two arguments of type `atom` make the intent to be sent. The next five constructors are for reading/writing from/to a provider, evaluate, fork, choice, result and void, respectively.

B. OPENING AND SUBSTITUTION OPERATIONS

The major advantage of locally nameless representation is that all alpha-equivalent terms now have unique representation and thus issues with alpha-equivalence and variable capture are avoided. To formally implement locally nameless representation, some basic operations are required. The function `fv` (free variables) is recursively defined in Fig. 8 which collects a set of names that are free in terms. There is no free variable in term `bvar` and `void` and singleton sets of names are returned for the term `fvar`, `ret`, `read`, and `result` where the name in the argument of each term is the only member of the set returned. For all other terms, the set of free names is the union of names in all sub-terms.

```
Fixpoint open_rec (k : nat) (u : term) (t : term) {struct t} : term :=
  match t with
    | bvar v ⇒ if k === v then u else (bvar v)
    | fvar v ⇒ fvar v
    | lam t' ⇒ lam (open_rec k u t')
    | call x1 x2 ⇒ call x1 x2
    | ret x ⇒ ret x
    | bind x1 x2 t' ⇒ bind x1 x2 (open_rec k u t')
    | reg p t1 ⇒ reg p (open_rec k u t1)
    | send p x1 x2 ⇒ send p x1 x2
    | read x ⇒ read x
    | write x1 x2 ⇒ write x1 x2
    | lett t1 t2 ⇒ lett (open_rec k u t1) (open_rec k u t2)
    | fork t' ⇒ fork (open_rec k u t')
    | choice t1 t2 ⇒ choice (open_rec k u t1) (open_rec k u t2)
    | result x ⇒ result x
    | void ⇒ void
  end.
```

FIGURE 9. Definition of function for opening terms.

An important operation is opening a term (abstractions such as lambda terms and let expressions) where a bound variable in a term is instantiated (replaced) with a term.

```
1 | Definition open t u := open_rec 0 u t.
```

FIGURE 10. Definition of function for opening terms with one binder.

This operation is used to pass through a binder and turn a bound variable into a free variable. The function `open_rec` (Fig. 9) takes an index representing a bound variable and two terms and replaces the index (bound variable) in one term (third argument) with another term (second argument). The function has no effect on terms without bound variables (such as `fvar`, `call`, `ret`, `read`, `write`, `result` and `void`). The function is defined general that opens up terms deep into multiple binders. However, this complicates the proof and is, therefore, limited to zero index by another definition in Fig. 10.

```
1  Fixpoint subst (z : atom) (u : term) (t : term) {struct t} : term :=
2  match t with
3    | bvar v ⇒ bvar v
4    | fvar v ⇒ if v == z then u else (fvar v)
5    | lam t' ⇒ lam (subst z u t')
6    | call x1 x2 ⇒ call x1 x2
7    | ret x ⇒ ret x
8    | bind x1 x2 t' ⇒ bind x1 x2 (subst z u t')
9    | reg p t' ⇒ reg p (subst z u t')
10   | send p x1 x2 ⇒ send p x1 x2
11   | read x ⇒ read x
12   | write x1 x2 ⇒ write x1 x2
13   | lett t1 t2 ⇒ lett (subst z u t1) (subst z u t2)
14   | fork t' ⇒ fork (subst z u t')
15   | choice t1 t2 ⇒ choice (subst z u t1) (subst z u t2)
16   | result x ⇒ result x
17   | void ⇒ void
18 end.
```

FIGURE 11. Definition of function for variable substitution.

To ease proofs, the free variable substitution (Fig. 11) operation is required, which replaces a free variable (first argument) in a term (third argument) with another term (second argument). An environment `env` is defined as a list of name and type pairs each binding a variable with a type. An environment is well-formed (`ok`) if there is no duplicated name (each atom is bounded at most once).

C. FORMALIZING TYPE CHECKER

To capture secure data flows within mobile device and enforce best practices in applications development, a type checker is defined in the theorem prover. The type checker can be defined either as an inductive type or as a recursive function. Inductive definitions in theorem prover are favorable when formal proofs are carried out while recursive definitions are good for executing programs. As binders in programming languages complicates formal reasoning in theorem provers, the focus should be to facilitate the more challenging part: the formal proofs in theorem prover. This motivates towards inductive definition of Android type checker in Coq.

The type system for well-typed code in [13] is inductively defined as a relation `typing` (Fig. 12). The relation checks

```
1  Inductive typing : env → term → Ty → Prop :=
2   | typing_var : forall E (v : atom) T,
3        ok E →
4        binds v T E →
5        typing E (fvar v) T
6   | typing_read : forall E x,
7       (typing E (result x) (Ty_sub (ty_pro READ WRITE))) →
8       (typing E (read x) (Ty_sub (ty_any READ WRITE)) )
9   | typing_write : forall E n v,
10      (typing E (result n) (Ty_sub (ty_pro READ WRITE))) →
11      (typing E (result v) (Ty_sub (ty_any READ WRITE))) →
12      (typing E (write n v) (Ty_sub (ty_any BOTTOM TOP)))
13  | typing_reg : forall L E t tau T T',
14      (forall x:atom, x 'notin' L →
15      (typing ( (x, (Ty_sub tau)) :: E) (open t x) T)) →
16      (T' = (ty_rvr SEND tau T)) →
17      (typing E (reg SEND t) (Ty_sub T') )
18  | typing_send : forall E n tau T v,
19      (typing E (result n) (Ty_sub (ty_rvr SEND tau T))) →
20      (typing E (result v) (Ty_sub tau)) →
21      (typing E (send RECEIVE n v) T)
22  | typing_lett : forall L T1 T2 t1 t2 E,
23        typing E t1 T1 →
24        (forall x:atom, x 'notin' L →
25        typing ( (x, T1) :: E) (open t2 x) T2) →
26        typing E (lett t1 t2) T2
27  | typing_fork : forall E t tau,
28        (typing E t (Ty_sub tau)) →
29        (typing E (fork t) (Ty_sub (ty_any BOTTOM TOP)))
30  | typing_choice : forall E t T t',
31        (typing E t T) →
32        (typing E t' T ) →
33        (typing E (choice t t') T)
34  | typing_val_hyp : forall E (x : atom) T,
35        ok E →
36        binds x T E →
37        typing E (result x) T
38  | typing_val_void : forall E,
39     (typing E void (Ty_sub (ty_any BOTTOM TOP)))
40  | typing_call : forall E n tau tau' v,
41     (typing E (result n) (Ty_sub (ty_act CALL tau ty_dnc tau'))) →
42     (typing E (result v) (Ty_sub tau)) →
43     (typing E (call n v) (Ty_sub tau'))
44  | typing_return : forall E v tau,
45     (typing E (result v) (Ty_sub tau)) →
46     (typing E (ret v) (Ty_stack ty_dnc tau)).
```

FIGURE 12. Definition of type system.

that a term is well-typed (follows certain rules) in the environment. There are twelve typing rules (each begin with a vertical line |) for twelve terms. These terms are selected in the rules based on their relevance for enforcing best practices. In addition to a rule for free variables (`typing_var`, lines 2–5, Fig. 12), there is a constructor for each rule of the type system in [13]. The predicate `ok` ensures the environment does not have duplicate bindings and the relation `binds` binds the variable `x` with type `T` in the environment `E`.

To read data from a content provider (lines 6–8, Fig. 12) with permissions `READ`/`WRITE` in its context, the reader component must have the same `READ`/`WRITE` permissions (*data can flow from contexts with at most permission* `WRITE` *to contexts with at least permission* `READ` [13]). Similarly, a trusted component (with highest integrity level `⊤`) can write data (lines 9–12) with the `WRITE` label to a provider with specified permission `WRITE`. The rule `typing_reg` (lines 13–17) registers a new receiver and sets it with permission `SEND` that will be required in the context to send data to

it. This rule deals with binders and cofinite quantification [15] is introduced here. The body of term `t` is opened up with a free variable named `x` where the set `L` is some finite set of names chosen. A broadcast receiver can receive intents only if it has the permission `RECEIVE` in its context. The rest of the rules are either straight forward or explained in [13]. Note that, the rule `typing_lett` and `typing_reg` deals with binders and hence are formalized using cofinite quantification.

The type checker, as defined earlier, can be applied to any term (program) and their correctness can be checked in presence of other conditions. Just as an example, consider an activity component of the phone application requests (read) a contact `x` from the content provider of the contacts application. This is modeled as program `read x` with security labels `READ`/`WRITE` for the data `x` and content provider context. The *best practice* enforces the security policy described in Section I: the data from a content provider with a label can only be read by a component with a label at least as high as the content provider. The correctness of the term in accordance to the best practice is stated as a lemma in Fig. 13 and proved in proof assistant Coq.[3] Conformance of other terms with the security policy can similarly be stated and proved mechanically. Readers are advised to refer to [12] for other encoded examples in the type system defined in Fig. 3 and formalized in 12.

```
1 | Lemma checking_read: forall E x,
2 |   (typing E (result x) (Ty_sub (ty_pro READ WRITE)) ) ->
3 |   (typing E (read x) (Ty_sub (ty_any READ WRITE)) ).
```

FIGURE 13. Type checking term `read x`.

IV. PROOF OF SOUNDNESS PROPERTIES

The formal definition of syntax of language to describe Android applications, basic operation, and Android type system in theorem prover altogether build language-based security on Android systems. To demonstrate that the formal development can effectively be used to formally reason about the type system and Android terms (applications), formal proof of properties of the type system are carried out in the Coq tool.

The type system in Fig. 12 establishes rules for well-formed (safe) code. To ensure that well-typed expressions indeed are correct (allows only safe data flows) and will never go wrong, type soundness properties of the type system itself needs to be satisfied. Proofs of two properties, weakening and strengthening, of the type system defined in Fig. 12 are carried out in theorem prover.

A. PROOF OF WEAKENING PROPERTY

The weakening property states that adding type bindings to the environment does not affect derivability. In other words, it states that if an expression is well-typed in an environment Γ, then it is well-typed after the environment is extended with a name bound to some type `U`. Formally,

$$\text{(R-WEAKENING)}\quad \frac{\Gamma \vdash e : T}{\Gamma, n : U \vdash e : T}$$

The same property is formalized as a lemma in Fig. 14, where `E` and `F` are non-empty environments, the operator `++` concatenates the two environments, and the predicate `ok` ensures the concatenation of two environments do not have double bindings.

```
1 | Lemma typing_weakening : forall E F e T,
2 |     typing E e T →
3 |     ok (F ++ E) →
4 |     typing (F ++ E) e T.
```

FIGURE 14. Weakening property lemma.

To prove this lemma, a more stronger property needs to be proved first. The property is strong in the sense that the property holds for extension of the environment at the middle. This lemma is proved using induction on the relation `typing`. This stronger property, lemma `typing_weakening_strengthened` in Fig. 15, is then used (simple application) in the proof of lemma `typing_weakening` in Fig. 14.

```
1 | Lemma typing_weakening_strengthened : forall E F G t T,
2 |   typing (G ++ E) t T →
3 |   ok (G ++ F ++ E) →
4 |   typing (G ++ F ++ E) t T.
```

FIGURE 15. Strong weakening property lemma.

B. PROOF OF STRENGTHENING PROPERTY

Similarly, a more complex property strengthening states that removing type bindings preserves derivability as long as they are not present into the expression, which is type-checked. In other words, this property states that if a command is well-typed (correct) in an environment where a name `n` is bound to type `U`, then the command will still be correct if the name `n` is removed from the environment, provided that the name `n` is not part of the command. Formally, it is shown by the following inference rule:

$$\text{(R-STRENGTHENING)}\quad \frac{\Gamma, n : U \vdash e : T \quad n \notin \{freenamesofe\}}{\Gamma \vdash e : T}$$

The above rule is formalized in Coq as lemma in Fig. 16. The function `fv` returns the set of free names in term `e`. This lemma is proved using induction on the relation `typing`.

C. DISCUSSION

The first soundness property, weakening, of our type system asserts that if an Android application component is proved

[3] All the Coq definitions and proofs are available from the Github repository at https://github.com/wilstef/secdroid

```
Lemma Strengthening : forall e (n:atom) G U E T,
  n `notin` (fv e) →
  (typing (G ++ [(n, U)] ++ E) e T) →
  (typing (G ++ E) e T).
```

FIGURE 16. Strengthening property lemma.

to be data-flow safe, it remains safe even if a new application component or resource is added to the mobile device. Similarly, the strengthening property states that if an application component is independent from another application component, un-installation of the latter component does not invalidate the former provided it was valid prior to the un-installation.

The language-based security defined in previous sections captures, at an abstract level, how Android application components can securely communicate with each other using Android permissions. We carried out mathematical proof in Coq theorem prover, which confirms the correctness of the type system in itself. The security system was defined in the Coq theorem prover with a type checker program available that checks the syntax correctness of the security (type) system as well as provides foundation for other mathematical reasoning. The major advantage of formal analysis of systems using interactive proof assistant, such as Coq, is that they are powerful, expressive and the correctness of the proofs can be checked automatically using a computer.

V. RELATED WORK

Interactive proof assistants have been applied in the past to investigate Android systems security. In a recent work by one of the co-authors [24], they formally analyzed the security of Andriod inter-component communication based on intent messages. A formal model, dubbed as CrashSafe, was defined in Coq and used in formal verification of crash safety property of Android applications. CrashSafe is lightweight formal model of Android inter-component communication, however, it does not include Android permissions system. Even though, there are means to circumvent Android permissions [10], but it is still the main defence mechanism of Android system, if used carefully by applications developer. Shin *et al.* [25], [26] proposed formal model of Android permission mechanism and investigated it by specifying Android system elements, characterized security properties and proved that the system preserves the security properties. They specified the permission mechanism for Android system as a state machine and proved in Coq, the system is secure over the specified states and transitions. Their work focuses on the behavioral aspect of the framework rather than evaluating applications based on the Android permissions in order to exclude malicious ones. In a similar work, Betarte *et al.* [27], [28] developed a comprehensive formal specification of the permissions in Coq and verified several security properties. These formal models either do not address security issues related to Android permissions [24] or capture only specific security vulnerabilities in Android permissions systems [25]–[28]. The type checker developed in this paper, on the other hand, enforces best practices, which capture classes of security vulnerabilities. Furthermore, the enforcement mechanism in this paper is based on language-based security technique, which offers numerous advantages over other techniques [13], [23].

ScanDroid [12] extracts application specific security specifications from applications' manifests and applies data flow analysis to check data flow consistency with respect to the specifications. Felt *et al.* [6] found that about one-third of applications request permissions they normally do not need. They developed a tool Stowaway to detect *overprivilege* in Android applications by finding the maximum set of permissions required for an application and compared it with the set of permissions requested. Stowaway can determine if an application is overprivileged, however, it does not formally study the security implications of overprivilege. There are a number of other tools developed to reveal potential risks in Android system or improve their overall security. TaintDroid [29] demonstrates potential threats to phone users' data from third-party applications. Chin *et al.* developed ComDroid to improve Android applications security by detecting inter-application communication. A number of tools, such as DroidSafe [30], Horndroid [31] and Flowdroid [32] to name a few, have been proposed to analyze the data flow security of Android applications.

Locally nameless representation has been used in a number of other formal results using theorem prover Coq or Isabelle/HOL. A non-exhaustive list of research papers employing locally nameless representation is given in [15]. Using Coq theorem prover, Jia *et al.* [16] proved decidability and soundness of the type system of programming language AURA. Benton and Koutavas [17] introduced bisimulation for ν-calculus and formalized its metatheory in Coq. Garrigue [18] formalized a certified interpreter for the core ML using theorem prover Coq.

VI. CONCLUSIONS AND FUTURE WORK

The permission system in Android applications enables applications from different developers to share data and resources with each other. This mutual interaction between applications based on permissions may cause security consequences, such as threats to secrecy and integrity of data. To capture safe data flows between Android applications, programming language-based security on Android has been formalized in mechanical theorem prover Coq. The formal system defined in Coq can effectively be used to study correctness (data-flow safety) of Android applications, modeled as simple terms. Furthermore, the language-based technique formalized in theorem prover enables mechanical support for creating and checking proof of correctness of applications as well as the language-based technique itself.

The formal model demonstrates theorem prover Coq, locally nameless representation and language-based security technique all together can be used to analyze the security provided by the *model* of the permission system and correctness of the technique itself. To make it more practical,

a software tool translating real Android applications to the logical formulas of the model would be more interesting. Moreover, the proof of soundness properties may be extended to other properties such as type safety and subject reduction.

REFERENCES

[1] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Commun. Mobile Comput.*, vol. 13, no. 18, pp. 1587–1611, Dec. 2013.

[2] *Android Apps on Google Play*. Accessed: Jul. 15, 2018. [Online]. Available: https://play.google.com/store/apps?hl=en

[3] *Apple Apps Store*. Accessed: Jul. 15, 2018. [Online]. Available: https://itunes.apple.com/us/genre/ios/id36?mt=8

[4] *Android*. Accessed: Jul. 23, 2018. [Online]. Available: https://www.android.com/

[5] The Statistics Portal. *Global Mobile OS Market Share 2009–2017, by Quarter*. Accessed: Jul. 23, 2018. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 627–638.

[7] *Application Fundamentals*. Accessed: Jul. 23, 2018. [Online]. Available: https://developer.android.com/guide/components/fundamentals.html

[8] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security Privacy*, vol. 7, no. 1, pp. 50–57, Jan. 2009.

[9] T. Vennon and D. Stroop, "Android market: Threat analysis of the Android market," SMobile Syst., Columbus, OH, USA, Tech. Rep., 2010. Accessed: Dec. 20, 2018. [Online]. Available: https://www.slideshare.net/yannriviere/android-market-threat-analysis-2210v1

[10] C. Orthacker *et al.*, "Android security permissions–can we trust them?" in *Proc. Int. Conf. Secur. Privacy Mobile Inf. Commun. Syst.*, 2011, pp. 40–51.

[11] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. USENIX Secur. Symp.*, vol. 6, 2011, pp. 12–16.

[12] A. Chaudhuri, A. Fuchs, and J. Foster, "SCanDroid: Automated security certification of Android applications," Univ. Maryland, College Park, MD, USA, Tech. Rep. CS-TR-4991, 2009.

[13] A. Chaudhuri, "Language-based security on Android," in *Proc. ACM SIGPLAN 4th Workshop Program. Lang. Anal. Secur.*, 2009, pp. 1–7.

[14] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich, "Engineering formal metatheory," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 3–15, 2008.

[15] A. Charguéraud, "The locally nameless representation," *J. Automated Reasoning*, vol. 49, no. 3, pp. 363–408, 2012.

[16] L. Jia *et al.*, "Aura: A programming language for authorization and audit," *ACM SIGPLAN Notices*, vol. 43, no. 9, pp. 27–38, 2008.

[17] N. Benton and V. Koutavas, "A mechanized bisimulation for the nu-calculus," Tech. Rep. MSR-TR-2008-129, 2008.

[18] J. Garrigue, "A certified implementation of ML with structural polymorphism," in *Proc. Asian Symp. Program. Lang. Syst.*, 2010, pp. 360–375.

[19] W. M. van Cuijk, "Enforcing a fine-grained network policy in Android," M.S. thesis, Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2011.

[20] INRIA. *The Coq Proof Assistant*. Accessed: Jul. 24, 2017. [Online]. Available: https://coq.inria.fr/

[21] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283. Berlin, Germany: Springer-Verlag, 2002.

[22] A. Sabelfeld, "Language-based information security," in *Proc. Found. Comput. Secur.*, 2003, p. 99.

[23] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," in *Informatics*. New York, NY, USA: Springer, 2001, pp. 86–101.

[24] W. Khan *et al.*, "CrashSafe: A formal model for proving crash-safety of Android applications," *Hum.-Centric Comput. Inf. Sci.*, vol. 8, no. 1, p. 27, 2018.

[25] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "Towards formal analysis of the permission-based security model for Android," in *Proc. 5th Int. Conf. Wireless Mobile Commun. (ICWMC)*, Aug. 2009, pp. 87–92.

[26] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A formal model to analyze the permission authorization and enforcement in the Android framework," in *Proc. IEEE 2nd Int. Conf. Social Comput. (SocialCom)*, Aug. 2010, pp. 944–951.

[27] G. Betarte, J. D. Campo, C. Luna, and A. Romano, "Verifying Android's permission model," in *Proc. Int. Colloq. Theor. Aspects Comput.*, 2015, pp. 485–504.

[28] G. Betarte, J. Campo, M. Cristiá, F. Gorostiaga, C. Luna, and C. Sanz, "Towards formal model-based analysis and testing of Android's security mechanisms," in *Proc. 43rd Latin Amer. Comput. Conf. (CLEI)*, Sep. 2017, pp. 1–10.

[29] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, 2010.

[30] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in droidsafe," in *Proc. NDSS*, 2015, pp. 1–16.

[31] S. Calzavara, I. Grishchenko, and M. Maffei, "Horndroid: Practical and sound static analysis of android applications by smt solving," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Mar. 2016, pp. 47–62.

[32] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

WILAYAT KHAN received the B.Sc. degree in engineering from the University of Engineering and Technology, Peshawar, Pakistan, in 2007, and the M.S. degree in IT from Kungliga Tekniska Hogskolan, Sweden, in 2009. He is currently pursuing the Ph.D. degree with the University of Venice, Italy. He was a Lecturer with the COMSATS Institute of IT, Pakistan, until 2011. During his Ph.D. degree, he was an Exchange Researcher with KU Leuven University, Belgium, for five months. In 2014, he attended the prestigious Oregon Programming Language Summer School, University of Oregon, USA. He was a Research Fellow with Nanyang Technological University, Singapore. He is currently an Assistant Professor with COMSATS University Islamabad at Wah Campus, Pakistan. He has published a number of research papers in reputed journals and conference proceedings. He is the author of hardware description language VeriFormal, Chrome extension CookiExt, and formal Boolean equivalence tool BECheck. He is working on using formal methods to investigate computer systems. In particular, he is interested in using theorem provers to build formal models of computer hardware and software systems and prove their security and reliability properties.

MUHAMMAD KAMRAN received the M.S. and Ph.D. degrees in computer science from the National University of Computer and Emerging Sciences, Islamabad, Pakistan, in 2008 and 2012, respectively. He is currently an Assistant Professor with the Department of Computer Science, COMSATS University Islamabad at Wah Campus, Wah Cantonment, Pakistan. His research interests include machine learning, evolutionary computation techniques, information security, health informatics, big data analytics, and decision support systems.

AAKASH AHMAD received the Ph.D. degree in software engineering from Dublin City University, Dublin, Ireland, in 2015. He was a Postdoctoral Researcher with the IT University of Copenhagen and a Software Engineer with Elixir Technologies. He is serving as an Assistant Professor with the College of Computer Science and Engineering, University of Hail, Ha'il, Saudi Arabia. His research interests include software architecture and software patterns for mobile and cloud computing systems.

FARRUKH ASLAM KHAN received the M.S. degree in computer system engineering from the GIK Institute of Engineering Sciences and Technology, Pakistan, in 2003, and the Ph.D. degree in computer engineering from Jeju National University, South Korea, in 2007. He also received professional trainings from the Massachusetts Institute of Technology, New York University, IBM, and other professional institutions. He has successfully supervised four Ph.D. students and 16 M.S. thesis students. Several M.S. and Ph.D. students are currently working under his supervision. He is currently an Associate Professor with the Center of Excellence in Information Assurance, King Saud University, Riyadh, Saudi Arabia. He is also the Founding Director of the Wireless Networking and Security Research Group, National University of Computer and Emerging Sciences, Islamabad, Pakistan. He has over 80 publications in refereed international journals and conferences. His research interests include cyber security, body sensor networks and eHealth, bio-inspired and evolutionary computation, and the Internet of Things. He is a Senior Member of the IEEE. He currently serves as an Associate Editor for prestigious international journals, including the IEEE Access, *PLOS ONE*, *Neurocomputing* (Elsevier), *Ad Hoc and Sensor Wireless Networks*, *KSII Transactions on Internet and Information Systems*, *Human-Centric Computing and Information Sciences* (Springer), and *Complex & Intelligent Systems* (Springer). He is on the panel of reviewers of over 30 reputed international journals and numerous international conferences. He has also co-organized several international conferences and workshops.

ABDELOUAHID DERHAB received the B.Sc., M.S., and Ph.D. degrees in computer science from the University of Sciences and Technology Houari Boumediene, Algiers, in 2001, 2003, and 2007, respectively. He was a full-time Researcher with the CERIST Research Center, Algeria, from 2002 to 2012. He is currently an Associate Professor with the Center of Excellence in Information Assurance, King Saud University, Riyadh, Saudi Arabia. His research interests include network security, intrusion detection systems, malware analysis, mobile security, and mobile networks.