An Automatically Verified Prototype of the Android Permissions System

Maximiliano Cristiá 1 · Guido De Luca 2 · Carlos Luna 3

Received: 21 September 2022 / Accepted: 11 April 2023 / Published online: 12 May 2023 © The Author(s), under exclusive licence to Springer Nature B.V. 2023

Abstract
In a previous work we presented formal specifications of idealized formulations of the permission model of Android in the Coq proof assistant. This formal development is about 23 KLOC of Coq code, including proofs. In this work the Coq model is encoded in \{log\} (‘setlog’) — a satisfiability solver and a constraint logic programming language — which is then used to automatically discharge most of the proofs performed in Coq. We show how the Coq model is encoded in \{log\} and how automated proofs are performed. The resulting \{log\} model is an automatically verified executable prototype of the Android permissions system. Detailed data on the empirical evaluation resulting after executing all the proofs in \{log\} is provided. The integration of Coq and \{log\} as to provide a framework featuring automated proof and prototype generation is discussed.

Keywords Android · Coq · \{log\} · Security properties · Automated proof

1 Introduction

Mobile devices have become an integral part of how people perform tasks in their work and personal lives. The benefits of using mobile devices, however, are sometimes offset by increasing security risks.

Android [43] is an open platform for mobile devices—developed by the Open Handset Alliance led by Google Inc.—that captures more than 85% of the total market-share [35].

---

Maximiliano Cristiá, Guido De Luca and Carlos Luna have contributed equally to this work.

Maximiliano Cristiá
cristia@cifasis-conicet.gov.ar

Guido De Luca
gdeluca@dcc.fcen.uba.ar

Carlos Luna
cluna@fing.edu.uy

1 Universidad Nacional de Rosario and CIFASIS, Rosario, Argentina
2 Universidad Nacional de Rosario, Rosario, Argentina
3 Universidad de la República, Montevideo, Uruguay
Android offers many critical applications in terms of privacy. In order to offer their users security, Android relies on a multi-party consensus system where user, OS and application must be all in favor of performing a task. Android implements security mechanisms at both OS and application level. In particular, Android behaves like a multi-user Linux system. Application-level access control is implemented by an Inter-Component Communication reference monitor that enforces mandatory access control policies regulating access among applications and components. Application security is built primarily upon a system of permissions, which specify restrictions on the operations a particular process can perform.

In previous works two of the authors present formal specifications in the Coq proof assistant [6, 52] of idealized formulations of different versions of the permission model of Android [7–9]. These works formulate and demonstrate, in a non-automatic way, a set of properties of the Android security model. In particular, De Luca and Luna [30] formalize the Android permissions system introduced in versions Nougat, Oreo, Pie and 10. The formal development is about 23 KLOC of Coq code, proofs included.

In this work the Coq model is encoded in \{log\} [45]—a satisfiability solver and a constraint logic programming language— which is then used to automatically discharge most of the proofs performed in Coq. In this way much of the manual, expert work needed to prove properties in Coq can be avoided. In particular, Cristiá, assisted by De Luca and Luna, encoded in \{log\} the Coq model developed by his coauthors. Hence, this paper shows in detail how different elements and properties of the Coq model can be encoded and proved in \{log\}. The resulting \{log\} model is an executable prototype. Additionally, detailed data on the empirical evaluation resulting after executing all the proofs in \{log\} is provided. Finally, the integration of Coq and \{log\} as to provide a framework with automated proof and prototype generation is briefly discussed.

We believe that combining \{log\} with a powerful proof assistant such as Coq can produce a tool which would reduce to a minimum the manual effort needed during the formal verification of systems such as the Android permissions system. Hence, possible integration strategies are discussed in the paper.

**Organization of the paper** The rest of the paper is organized as follows. Section 2 provides some background on the Android permission system, its Coq model and \{log\}. Next, Sect. 3 shows how different elements of the Coq model are encoded in the \{log\} language, whereas Sect. 4 shows how the properties proved true of the Coq model can be encoded as \{log\} satisfiability queries. Section 4.3 provides detailed data on the empirical evaluation resulting after executing all the \{log\} queries. Section 5 discusses a possible integration strategy between Coq and \{log\} and considers related work. Finally, Sect. 6 concludes with a summary of our contributions and directions for future work.

## 2 Background

This section contains some background on the Android permissions system (2.1); the Coq model developed by us (2.2); and the \{log\} constraint solver (2.3). Readers may skip any of these subsections if they are familiar with those topics.

### 2.1 The Android Permissions System

The Android security model takes advantage of the fact that the entire platform is built upon a Linux kernel and relies primarily on a user-based protection to identify and isolate the
resources from each application. In other words, each Android application will have a unique user ID (UID), with restricted permissions, and will run its own process in a private virtual machine. This means that by default, applications will not be able to interact with each other and they will also have limited access to OS resources.

In the Android permissions system, every resource is protected by unique tags or names, called permissions, that applications must have been granted before being able to interact with the resource. Permissions can be defined by applications, for the sake of self-protection; and are predefined by Android so applications can get access to system resources, such as the camera or other kind of sensors. Applications must list the permissions they need in the so-called application manifest. Permissions are also associated to a protection level, depending on how critical the resource that they are securing is. The protection level also determines if the permission will be automatically granted upon installation or if user consent will be required at runtime. Android defines three protection levels: normal, signature and dangerous.

Permissions related to the same device capability, are grouped into permission groups. For example, reading and editing contacts are two different permissions. However, when an application requests one of these permissions for the first time, the user will be asked to authorize the “contacts group”. If the user accepts, then the behavior goes as follows:

1. The requested permission is granted and the system is notified that the user authorized the “contacts group”.
2. As soon as the application requests another permission from the same group, the system will automatically grant it, without informing the user.

2.2 A Verified Coq Model

As we have said, the Android permissions system has been modeled using the Coq proof assistant [7–9, 30]. Coq is a formal proof system based on the calculus of constructions [18, 44] logic that allows writing formal specifications and interactively generating machine-checked proofs of theorems. It also provides a (dependently typed) functional programming language that can be used to write executable algorithms. The Coq environment also provides program extraction towards languages like Ocaml and Haskell for execution of certified algorithms [39]. Proofs in Coq are essentially manual and interactive.

The Coq model of the Android permissions system follows the structure of an abstract state machine. In this sense, the model defines a tuple representing the set of states of that machine (Sect. 2.2.1) and a number of state transitions (called operations) given by means of their pre- and post-conditions (Sect. 2.2.2). The state of the machine contains both dynamic and static data of the system. In turn, the operations correspond to the actions that either the user, applications or Android can perform concerning the permission system. The security properties that have been proved true of the the model are discussed in Sect. 2.2.3.

The source code of the Coq model and the proof scripts can be found in a Github repository [40]. File names mentioned in this section refer to that repository. Some expressions used as examples in this work (namely defPermsForApp and mapping), relies on Coq’s sectioning mechanism. In each section Coq permits to declare parameters that work as implicit arguments to all of the definitions inside the section. For example, inside the section that contains defPermsForApp, we declare s : System as a parameter. However, in this paper, the declaration of parameters is omitted for brevity. Then, for instance in Fig. 2, we can see that defPermsForApp depends on s which corresponds to the section parameter. For further reference, please see the source code of the Coq project.
2.2.1 The State Space

In Fig. 1, we show a fragment of Coq code corresponding to the state definition, called System. As can be seen, System is the union of State and Environment. The former contains information such as which applications are currently installed (apps), which applications have been executed at least once (alreadyVerified), which groups of permissions were already authorized (grantedPermGroups) and which permissions were granted to each application (perms). In turn, Environment takes care of tracking information that does not change once an application is installed. For example, it contains a mapping between each application and its manifest (manifest), and a mapping between each application and the permissions defined by it (defPerms). In this way System sums up a total of 13 components representing the whole state of the model. The definitions of more detailed types are not included, for the sake of keeping the presentation brief. The types starting with lowercase correspond to basic parameters modelled as Set in Coq, whereas those starting with a capital letter were defined by more complex Records. From now on, whenever we mention the model’s states, we will be referring to elements of type System, unless stated otherwise.
The model defines a notion of valid state that captures several well-formedness conditions. This is so to avoid reasoning over some values of System that make no sense either in a real-world scenario or in the Coq representation of the system. For example, the valid state predicate forces the permissions defined by applications to be uniquely identified. In other words, we do not allow two different user-defined permissions having the same ID. This Coq predicate, called notDupPerm, is shown in Fig. 2. Note that without notDupPerm, an application could get access to a permission that is protecting some resource just by defining a permission of its own with the same ID.

The valid state predicate also forces that every mapping in the state is a partial function. Figure 3 shows a fragment of this predicate, called allMapsCorrect. The fixpoint declaration map_correct in line 7 is a recursive function that returns true only if there are no repeated elements in the domain of the map. The definition of the valid state property is the result of conjoining 13 different predicates, which are defined within 200 lines of code (file Estado.v).

2.2.2 Operations as State Transitions

The state transitions of the model, called “actions” or “operations”, cover the main functionalities of the Android permission system. As an example, Fig. 4 shows the semantics of the operation grantAuto, which is responsible for granting a permission to an application that already has an authorization for automatically granting permissions from a certain group. This action is executed whenever a dangerous permission is granted without explicit consent from the user.

The precondition (pre_grantAuto) establishes that the permission \( p \) is listed on the application’s manifest (and this manifest, of course, is required to exist). Regarding \( p \), it is also required that it is defined either by the user or the system, that its level is dangerous and that it has not been already granted to \( \text{app} \). Note that the latter is required because lists (instead of sets) are used to keep track of the already granted permissions (see more on sets and lists in Coq in Remark 2 below). Last but not least, the precondition of this action also requires that \( p \) belongs to a group \( g \) that the user has previously authorized for automatic permission granting.

---

1. **Definition allMapsCorrect**: Prop :=
   2. map_correct (manifest (environment s)) \
   3. map_correct (cert (environment s)) \
   4. map_correct (defPerms (environment s)) \
   5. map_correct (grantedPermGroups (state s)) \
   6. ...

7. **Fixpoint map_correct (mp: mapping)**: Prop :=
   8. match mp with
   9.   | nil => True
   10. | (a::rest) => ¬(In (item_index a) (map_getKeys rest)) \
   11.    map_correct rest
   12. end.

**Fig. 3** Fragment of allMapsCorrect
Definition grantAuto (p:Perm) (a:idApp) (s s’:System) : Prop :=
  pre_grantAuto p a s \ post_grantAuto p a s s’.

Definition pre_grantAuto (p: Perm) (a: idApp) (s: System) : Prop :=
  (exists m:Manifest, isManifestOfApp a m s \ In p (use m)) \ /
  (isSystemPerm p \ usuDefPerm p s) \ /
  ~(exists 1Perm:list Perm, 
    map_apply (perms (state s)) a = Value 1Perm \ In p 1Perm) \ /
  pl p = dangerous \ /
  (exists (g: idGrp) (1Group: list idGrp), 
    maybeGrp p = Some g \ /
    map_apply (grantedPermGroups (state s)) a = Value 1Group \ /
    In g 1Group).

Definition post_grantAuto (p:Perm) (a:idApp) (s s’:System) : Prop :=
  grantPerm a p s s’ \ /
  (environment s) = (environment s’) \ /
  (apps (state s)) = (apps (state s’)) \ /
  ...  (* Other state components remain the same *)

Definition grantPerm (a:idApp) (p:Perm) (s s’:System) : Prop :=
  (forall (a’:idApp) (1Perm:list Perm), 
    map_apply (perms (state s)) a’ = Value 1Perm ->
    exists 1Perm’:list Perm, 
    map_apply (perms (state s’)) a’ = Value 1Perm’ ->
    forall p’:Perm, In p’ 1Perm’ -> In p’ 1Perm’) \ /
  (forall (a’:idApp) (1Perm’:list Perm), 
    map_apply (perms (state s)) a’ = Value 1Perm’ ->
    exists 1Perm:list Perm, 
    map_apply (perms (state s’)) a’ = Value 1Perm \ /
    forall p’:Perm, 
    In p’ 1Perm’ -> ~(In p’ 1Perm’ -> (a=a’ \ p=p’)) \ /
    (exists 1Perm’:list Perm, 
    map_apply (perms (state s’)) a = Value 1Perm’ \ /
    In p 1Perm’) \ /
    map_correct (perms (state s’)).

Fig. 4 Semantics of grantAuto in Coq

The postcondition (post_grantAuto) basically adds p to (perms a) in the new state s’. This behavior is handled by the auxiliary predicate (grantPerm a p s s’). The rest of the components of the state remain the same.

A non-extensive list of other actions that can be performed in the model is: i) grant, which is the complementary operation of grantAuto, since it represents a permission being granted with explicit consent from the user; ii) revoke or revokeGroup, to remove an ungrouped permission or all of the permission from a group, respectively; iii) hasPermission, to check whether an application has a certain permission or not at a given moment. The model supports 22 operations in total, some of them being considerably more complex than grantAuto. The definition of the formal semantics of these operations comprises around 1,400 lines of Coq code (file Semantica.v).
We close this section with the following important observations.

**Remark 1** (State validity is invariant) We have demonstrated using Coq that each operation preserves valid states.

**Lemma 1** (Validity is invariant)
\[
\forall (s', a) : \exists (lGroup : list idGrp), \\
\text{mapapply} \ (\text{grantedPermGroups} \ ((\text{state} s))) \ a = \text{Value} \ lGroup \ \land \\
\text{in} \ g \ lGroup \rightarrow \\
\text{exec} \ s' \ (\text{grantAuto} \ p \ a) \ s' \ \text{ok}.
\]

**Remark 2** (Sets and lists in Coq) In Coq sets of type \(T\) can be encoded as \(T \rightarrow \text{Prop}\), i.e. functions from \(T\) onto \(\text{Prop}\). An evident consequence of this choice is that the resulting specification is not executable. In particular, the program extraction mechanism provided by Coq to extract programs from specifications cannot be used in this case. Hence, a specification encoding sets as \(T \rightarrow \text{Prop}\) should be refined into an specification encoding sets as lists, i.e. \(\text{list} \ T\), if a prototype is needed. In this case, the new specification must take care of repetitions and permutations. The Coq model of the Android permissions system was meant to produce a certified program [30]. Then, it uses \(\text{list} \ T\) to encode sets.

\[\square\]

### 2.2.3 Properties of the Model

De Luca, Luna and their colleagues use Coq [7–9, 30] to analyze properties of the Coq model. In particular, several security properties establish that the model provides protection against unauthorized access to the sensitive resources of a device running Android [7, 8]. Using the Coq specification, several lemmas were proved showing that the Android security model meets the so-called principle of least privilege, i.e. that “each application, by default, has access only to the components that it requires to do its work and no more” [2].

In a recent work [30], we present and discuss some properties about Android 10. In that work the focus is on safety-related properties concerning the changes introduced on the later versions of Android (mainly Oreo and 10) rather than on security issues.

Furthermore, using the Coq model, we precisely state the conditions that would help preventing the exploitation of some well-known vulnerabilities of the Android system [14, 47], like the unauthorized monitoring of information (eavesdropping) or inter-application communication (intent spoofing). They also prove that, under certain hypotheses, these attacks cannot be carried out [7, 8]. Some of these potentially dangerous behaviors may not be considered in the informal documentation of the platform.

A total of 14 properties have been analyzed with Coq. Next we comment on two of them.

Figure 5 shows the Coq definition of the property stating that the system automatically grants a dangerous permission only when the user had previously authorized the permission group that contains it. Note that in the expression defined in lines 5-7, it is claimed that the group of the permission is not granted to the application in the state \(s\). Therefore, line 8 concludes that the execution of \(\text{grantAuto}\) cannot be successful.
Fig. 6 Edge case scenario found for the automatic granting feature

In turn, Fig. 6 shows the Coq encoding of a scenario where the system is able to automatically grant a dangerous permission to an application even though the application has no other permission of the same group at the moment. This “vulnerable” state is reached when the user authorized that group at some point, then all the permissions of that group were removed but the ability of automatic granting was still kept by the system.

2.3 The \{log\} Constraint Solver

This paper aims at showing that \{log\} can be used as an effective automated prover for systems such as the Android permissions system. In this way much of the manual, expert work needed to prove properties in Coq can be avoided. Consider that proving all the properties of the Coq model described in Sect. 2.2 requires 18 KLOC of Coq proof commands.

\{log\} is a publicly available satisfiability solver and a declarative, set-based, constraint-based programming language implemented in Prolog [45]. \{log\} is deeply rooted in the work on Computable Set Theory [12, 13], combined with the ideas put forward by the set-based programming language SETL [48].

The automated proving power of \{log\} comes from the implementation of several decision procedures for different theories on the domain of finite sets, finite set relation algebra and linear integer arithmetic [19, 20, 22–24, 27, 31]. All these procedures are integrated into a single solver which constitutes the core part of the \{log\} tool. However, the tool admits formulas outside those decision procedures. For these formulas, termination can no longer be ensured. Several in-depth empirical evaluations provide evidence that \{log\} is able to solve non-trivial problems [19–21, 24, 28], including the security domain [25, 26].

\{log\} provides constraints encoding most of the set and relational operators used in set-based specification languages such as B [1] and Z [50]. For example: \texttt{un}(A, B, C) is a constraint interpreted as \(C = A \cup B\); \texttt{in} is interpreted as set membership (i.e., \(\in\)); \texttt{dom}(F, D) corresponds to \(\text{dom } F = D\); \texttt{subset}(A, B) to \(A \subseteq B\); \texttt{comp}(R, S, T) to \(T = R \circ S\) (i.e., relational composition); \texttt{applyTo}(F, X, Y) is a weak form of function application; and \texttt{pfun}(F) constrains \(F\) to be a (partial) function. The language supports some forms of sets such as the empty set (\(\emptyset\)) and extensional sets (\(\{t/A\}\), interpreted as \(\{t\} \cup A\)). Formulas in \{log\} are built in the usual way by using the propositional connectives (e.g., \&, or, \neg, implies), and restricted universal and existential quantifiers (\texttt{foreach} and \texttt{exists}, see Sect. 2.3.1).

\{log\} can be used as both a programming language and a satisfiability solver. Within certain limits, \{log\} code enjoys the formula-program duality.

Example 1 The following \{log\} predicate computes the maximum of a set:
dec_p_type(smax(set(int),int)).

smax(S,Max) :- Max in S & foreach(X in S, X <= Max & dec(X,int)).

The first line declares the type of $\text{smax}$: the first argument is a finite set of integers whereas
the second is an integer. The second line defines a clause (as in Prolog) whose body is a \{log\} formula making use of a restricted universal quantifier (RUQ). $\text{dec}$ declares the type of $X$.

$s\text{max}$ is a program, so we can execute it:

\{log\} => smax({3,5,1,7,4},Max).
Max = 7

But $\text{smax}$ is also a formula, so we can prove properties true of it by proving that their negations are unsatisfiable:

\{log\} => neg(smax(S,M) & Y <= M & smax({Y / S},K) implies M = K).
f\text{alse}

That is, \{log\} failed to find a finite set $S$ and integer numbers $Y, M$ and $K$ satisfying the
above formula. Furthermore, if we attempt to prove an unvalid property \{log\} returns a
counterexample:

\{log\} => neg(smax(S,M) & smax({Y / S},K) implies M = K).
S = {M/N}, K = Y
Constraint: foreach(X in N,M>=X), Y>=M, foreach(X in N,Y>=X), M neq Y

In \{log\}, we do not need a specification and a program; the same piece of code is both
the program and its specification. This duality is not the result of integrating two tools but
a consequence of the mathematical and computational models behind \{log\}. However, the
formula-program duality pays the price of reduced efficiency. For this reason, a \{log\} program
must be considered as a prototype of a real implementation.

2.3.1 Restricted Quantifiers

As we have said, \{log\} supports restricted quantifiers (RQ) called $\text{foreach}$ and $\text{exists}$. RQ have been used across the \{log\} encoding of the Coq model of the Android permissions
system and they have a rather complex structure. For these reasons here we present $\text{foreach}$ with more detail—$\text{exists}$ is used in the same way. $\text{foreach}$ is a restricted universal quantifier (RUQ). The most general form of $\text{foreach}$ is the following:

\text{foreach}(x \in A, [e_1, \ldots, e_n], \phi(x, e_1, \ldots, e_n), \psi(x, e_1, \ldots, e_n)) \quad (1)

where $x$ can be a variable or an ordered pair of two distinct variables; $e_1, \ldots, e_n$ are variables implicitly existentially quantified inside the $\text{foreach}$; $\phi$ is a \{log\} formula; and $\psi$ is a
conjunction of so-called functional predicates. A predicate $p$ of arity $n+1$ ($0 < n$) is a func-
tional predicate iff for each $x_1, \ldots, x_n$ there exists exactly one $y$ such that $p(x_1, \ldots, x_n, y)$
holds; $y$ is called the result of $p$. For instance, $\text{un}$, $\text{dom}$ and $\text{comp}$ are functional predicates. In a $\text{foreach}$, $e_1, \ldots, e_n$ must be the results of the functional predicates in $\psi$. The second
and last parameters are optional. The semantics of (1) is:

$\forall x (x \in A \implies (\exists e_1, \ldots, e_n(\psi(x, e_1, \ldots, e_n) \land \phi(x, e_1, \ldots, e_n))))$
The structure of the \texttt{foreach} predicate was designed to avoid the introduction of as many 'uncontrolled' existential variables as possible. The introduction of existential variables brings in the problem of negating the predicate in such a way that the result is a formula laying inside the decidable fragment. This structure proved to be expressive enough as to work with real-world problems \cite{21, 25, 27}.

Nested \texttt{foreach} can be written as follows:

\[
\text{foreach}([x \text{ in } A, y \text{ in } B], \phi) \equiv \text{foreach}(x \text{ in } A, \text{foreach}(y \text{ in } B, \phi))
\]

### 2.3.2 Negation in \{log\}

Negation in \{log\} has to be treated carefully. \texttt{neg} computes the \textit{propositional} negation of its argument. In particular, if \( \phi \) is an atomic constraint, \( \text{neg}(\phi) \) returns the corresponding negated constraint. For example, \( \text{neg}(x \text{ in } A \& z \text{ nin } C) \) becomes \( x \text{ nin } A \text{ or } z \text{ in } C \). However, the result of \texttt{neg} is not always correct because, in general, the negated formula may involve existentially quantified variables, whose negation calls into play (general) universal quantification that \{log\} cannot handle properly. Hence, there are cases where \{log\} users must manually compute the negation of some formulas. The same may happen for some logical connectives, such as \texttt{implies}, whose implementation uses the predicate \texttt{neg}:

\[ F \implies G \text{ is implemented in } \{log\} \text{ as } \text{neg}(F) \text{ or } G. \]

### 3 Encoding the Coq Model in \{log\}

In this section we show how different elements of the Coq model described in Sect. 2.2 are encoded in the \{log\} language. The complete \{log\} code of the Android permissions system is publicly available.\(^1\) A possible general integration strategy between Coq and \{log\} is discussed in Sect. 5.

Figure 7 shows the \{log\} encoding of the Coq code shown in Fig.4. Differently from the Coq code, the \{log\} code gathers all the three Coq definitions in a single \{log\} clause (\texttt{grantAuto}). This is so, for example, because the pre- and post-condition may access the

\(^1\) https://www.clpset.unipr.it/SETLOG/APPLICATIONS/android.zip.
same component of the state or may use the same external function. As \( \text{log} \) performs a sort of symbolic execution over this code, these double accesses or calls imply more computations thus making automated proof to take considerably longer. These differences between proof assistants and automated provers should be considered when the model is written for either of both. We will further discuss this in Sect. 5. Below we explain the encoding in more detail.

The first line declares the type of \texttt{grantAuto} whose head is given in line 2 and whose body is given from line 3 on. The first parameter of \texttt{grantAuto}, \texttt{SystemPerm}, is implicit in the Coq code as it is declared as a global parameter. \( \text{log} \) does not support implicit parameters so they have to be explicitly included in every clause where they are needed. Then, \( s \) corresponds to \( S, p \) to \( P, \text{app} \) to \( A \) and \( s' \) to \( S' \) (in \( \text{log} \) variables begin with a capital letter). For example, the \texttt{dec_p_type} declaration states that the type of \( S \) is \texttt{system} which is defined as a tuple. In this case, each field in the Coq record corresponds to a component in the tuple, as follows (see Fig. 1):

\[
\begin{align*}
\text{set(idApp)}, \quad & \% \text{apps} \\
\text{set(idApp)}, \quad & \% \text{alreadyVerified} \\
\text{rel(idApp,set(idGrp))}, \quad & \% \text{grantedPermGroups} \\
\ldots \ldots 
\end{align*}
\]

In order to access each component, we have defined predicates, named as the Coq fields, that use Prolog unification:

\[
\begin{align*}
\text{dec_p_type(apps(state,set(idApp))).} \\
\text{apps(S,Apps) :- S = [Apps,_2,_3,_4,_5,_6,_7,_8,_9].}
\end{align*}
\]

More importantly, the type of each component of \texttt{system} is different from (although equivalent, in some sense, to) the corresponding Coq type. For instance, the Coq type for \texttt{apps} is \texttt{list idApp}, where \texttt{idApp} is of type \texttt{Set}. In turn, in \( \text{log} \), the type is \texttt{set(idApp)}, where \texttt{idApp} is a basic type. According to Coq’s and \( \text{log} \)’s semantics, \texttt{idApp} means roughly the same in both systems: it is just a set of elements. Conversely, the semantics of \texttt{list idApp} and \texttt{set(idApp)} is quite different because the former means that \texttt{apps} is a list in Coq whereas the latter implies that it is a finite set in \( \text{log} \). Recall that in Coq \texttt{apps} is declared as a list although the model uses it as a set.

Similarly, the Coq type of \texttt{grantedPermGroups} is \texttt{mapping idApp (list idGrp)}, where \texttt{mapping: Set := list item and item} is the record \{\texttt{idx:index; inf:info}\}. That is, \texttt{grantedPermGroups} is basically a list of ordered pairs. However, as the name suggests, the intended usage is for \texttt{grantedPermGroups} to be a finite map or partial function. Furthermore, the range of the map is supposed to be composed of sets rather than lists. The Coq model includes a state invariant stating that this component should be a partial function (recall Fig. 3). Instead, the \( \text{log} \) type of \texttt{grantedPermGroups} is \texttt{rel(idApp,set(idGrp))}. That is, in \( \text{log} \) this component is a binary relation whose range are sets. Following the Coq model, the \( \text{log} \) model includes a similar state invariant constraining \texttt{grantedPermGroups} to be a partial function (see Sect. 4.1).

Returning to Fig. 7, lines 3 and 4 extract from \( S \) and \( S' \) the system components that are needed in the clause. Differently from Coq, in \( \text{log} \) we need new variables to access each component. For example, in Coq we can do \( \text{(state s)} \) to access the state of the system whereas in \( \text{log} \) we do \( \text{state(S,St)} \) where \( \text{St} \) is a new variable and then we use \( \text{St} \) to access the components of the state.
In line 5, we call \texttt{isManifestOfApp(A,M,S)} where \( M \) is a new variable implicitly existentially quantified. That is, the existential quantifier (exists \( m: \text{Manifest}, \ldots \)) in line 2 of Fig. 4 is not necessary in \{log\}. The code in lines 6 and 7 corresponds to the Coq code \( \texttt{(isSystemPerm p \ \lor \ \text{usrDefPerm p s})} \). In Coq \texttt{isSystemPerm} has type \( \text{Perm \rightarrow Prop} \).

Line 8 is interesting because it shows how some complex Coq predicates are encoded as \{log\} constraints. Indeed, line 8 corresponds to \( \sim \text{(exists lPerm:list Perm, \ldots)} \) in lines 5–6 of Fig. 4. That existential quantifier states that: i) \( a \) does not belong to the domain of \( (\text{perms (state s)}) \); or ii) if it does, then \( p \) is not in its image. In \{log\} i) is encoded as \( \text{comp}([A,A],\text{Perms},\{\}) \) where \( \text{comp} (R,S,T) \) is a constraint interpreted as \( T = R \circ S \) (where \( \circ \) is relational composition), and \([A,A]\) is an ordered pair. In turn, ii) is encoded as \( \text{applyTo} (\text{Perms}, A, PS) \) which is a constraint stating minimum conditions to apply a binary relation to a point. If \( \text{Perms} \) can be applied to \( A \), then it has some image \( \text{PS} \); and when \( \text{Perms} \) cannot be applied to \( A \), we take \( \text{PS} \) as the empty set. See that \( \text{PS} \) is used in line 11 to set the new state of the system.

Encoding predicates such as \( \sim \text{(exists lPerm:list Perm, \ldots)} \) as \{log\} constraints is not the same as putting those predicates behind Coq definitions. A Coq definition works at the syntactic level whereas \{log\} implements those constraints at the semantic level. Therefore, encoding complex predicates as constraints is not just a matter of increasing the readability of the model but of increasing \{log\}'s automated proving capabilities.

This is particularly noticeable in line 11 of Fig. 7 as it encodes \( \text{grantPerm} \) of Fig. 4. \{log\} defines \( \text{foplus} \) as follows:

\[
\text{foplus}(F,X,Y,G):= \\
\textsf{F} = \{[[X,Z]/H]\ \& \ \text{comp}([[X,X]],H,\{\})\ \& \ G = \{[X,Y]/H\} \text{ or} \\
\text{comp}([[X,X]],F,\{\})\ \& \ G = \{[X,Y]/F\}.
\]

(2)

That is, \( G \) is equal to \( F \) except in \( X \) where: if \( X \) is in the domain of \( F \) then \( Y \) becomes the new image of \( X \); if not, \([X,Y]\) is added. When \( F \) is a function, \( G \) is uniquely defined.

Hence, line 11 is equivalent to state that \( (\text{perms (state s)}) \) is equal to \( (\text{perms (state s')}) \) except in \( a \) where either \( (a,\{p\}) \) is added to \( (\text{perms (state s)}) \) or the new image of \( a \) is its old image plus \( \{p\} \). All this, in turn, is expressed in Coq with the quantified predicates of \( \text{grantPerm} \).

In the last line, \texttt{updateGrantAuto} updates the system as follows:

\[
\text{updateGrantAuto(S,P,S_):=} \\
S = [St,_,S2] \ \& \ St = [_1,_,2,_,3,_,4,_,5,_,6,_,7,_,8,_,9] \ \& \\
S_ = [St_,_,S2] \ \& \ St_ = [_1,_,2,_,3,P,_,5,_,6,_,7,_,8,_,9].
\]

That is, lines 11-12 of Fig. 7 correspond to \texttt{post_grantAuto} in Fig. 4.

\textbf{Remark 3} The \{log\} encoding of Android enjoys the formula-program duality, as explained in Sect. 2.3. That is, on one hand, clauses such as \texttt{grantAuto} can be executed, thus turning them into a sort of prototype API that can be used to program security-related scenarios. On the other hand, the same \{log\} code is an specification of the Android permissions system. As such, it is possible to use \{log\} to (automatically) prove properties true of the specification. This is the subject of the next section.
4 Encoding Security Properties in {log}

As we have explained in Sects. 2.2.1 and 2.2.3 the Coq model verifies several properties. These properties can be divided into two classes: invariance lemmas concerning the concept of valid state (Sect. 2.2.1), and security properties (Sect. 2.2.3). In the introduction we set as one of the main goals of this paper assessing {log} as an automated prover for the kind of properties used to validate the Coq model—having as a long term goal integrating {log} and Coq to optimize the proof process for certain classes of problems.

Hence, in this section we show how the properties proved true of the Coq model can be encoded as {log} satisfiability queries. Then, these queries can be executed against the {log} program described in Sect. 3. Every time we get a false answer we know that the query is unsatisfiable meaning that its negation is a theorem (see Example 1). If, on the contrary, we get a solution (counterexample) we know the query is satisfiable meaning that its negation is not a theorem (property) derivable from the program.

4.1 Valid State

In this section we show how two of the properties defining the set of valid states are encoded in {log}. We start with allMapsCorrect defined in Fig. 3. As can be seen, this predicate is the conjunction of several map_correct predicates. In turn, map_correct states that its argument is a partial function. In {log} pfun(F) constrains F to be a partial function. Hence, when in Coq we have map_correct (grantedPermGroups (state s)) in {log} we write:

```
dec_p_type(allMapsCorrect4(system)).
allMapsCorrect4(S) :-
dec(GR,rel(idApp,set(idGrp))) &
state(S,St) & grantedPermGroups(St,GR) & pfun(GR).
```

Note that in {log}, as well as in Coq, we use a combination of types and constraints to state the desired property. That is, dec(GR, rel(idApp, set(idGrp))) restricts the domain of GR to idApp and the range to set(idGrp), whereas pfun(GR) constrains GR to be a function. The dec predicate is enforced during type checking whereas the pfun constraint is enforced during constraint solving.

Now we turn our attention to predicate notDupPerm shown in Fig. 2. As can be seen, this predicate begins with a universal quantification over several variables of different types. The Coq predicate then goes to restrict the quantified variables to belong to different sets. For instance, a and l must verify defPermsForApp a l. In turn, defPermsForApp is divided into two cases; let’s analyze the first one. In this case l is the image of a through (defPerms (environment s)). Then, in a set-based notation such as {log} this is equivalent to:

```
environment(S,E) & defPerms(E,DP) & [A,L] in DP
```

where S corresponds to s, and E and DP are new variables. Hence, in {log}, instead of quantifying over types, we can use a RUQ (cf. Example 1 and Sect. 2.3.1). In other words,

---

2 Some dec predicates are avoided for readability.

3 In Coq they are, formally, lists.
we turn Coq’s for all (a:idApp) (l:list Perm),... into foreach([A,L] in DP, ...). The second case of defPermsForApp can be treated in a similar way because in this case a and l are components of sysapp which in turn is an element of the set (systemImage (environment s)). The quantification over p and p’ will be discussed shortly.

Therefore, as a general rule, we turn Coq’s universal quantifications into \{log\}’s RUQ.

Finally, in order to encode notDupPerm in \{log\} we define three clauses corresponding to the result of distributing the two cases of defPermsForApp into the rest of the predicate. That is, in \{log\}, we have:

- notDupPerm1 where a, a’, 1 and 1’ are quantified over defPerms.
- notDupPerm2 where a, a’, 1 and 1’ are quantified over systemImage.
- notDupPerm3 where a and l are quantified over defPerms, whereas a’ and 1’ are quantified over systemImage.

As an example we show notDupPerm3:

\[
\begin{align*}
\text{dec_p_type(notDupPerm3(system)).} \\
\text{notDupPerm3(S) :-} \\
\text{environment(S,E) & defPerms(E,DP) & systemImage(E,SS) &} \\
\text{foreach([[A1,L1] in DP, A2 in SS], [ID2,L2],} \\
\text{foreach([P1 in L1, P2 in L2], [IP1,IP2],} \\
\text{IP1 = IP2 implies P1 = P2 & A1 = ID2,} \\
\text{idP(P1,IP1) & idP(P2,IP2)} \\
\text{),} \\
\text{idSI(A2,ID2) & defPermsSI(A2,L2)} \\
\end{align*}
\]

As can be seen, the universal quantification over p and p’ present in the Coq model becomes the inner formula of the outermost RUQ in \{log\}. For instance, L1 quantifies over the range of DP and then P1 quantifies over L1.

As explained in Sect. 2.2.1, the definition of valid state is given in terms of a state invariant (see Lemma 1). In \{log\}, instead of proving that the encoding of valid_state is a state invariant, we prove that each of the properties included in it is a state invariant. In other words, we prove the encoding in \{log\} of:

\[
I_1(s) \land s \xrightarrow{a} s' \rightarrow I_1(s')
\]

\[
.........................
\]

\[
I_n(s) \land s \xrightarrow{a} s' \rightarrow I_n(s')
\]

for all \(a : \text{Action}\) where valid_state(s) \(\equiv I_1(s) \land \cdots \land I_n(s)\). This is so because attempting to prove Lemma 1 may make \{log\} to incur in a lengthy computation due to the presence of unnecessary hypothesis. For example, if in order to prove \(I_k(s')\) the only necessary hypothesis are \(I_j(s)\) and \(a\), the presence of \(I_j(s)\) \((j \neq k)\) may make \{log\} to attempt to prove the lemma by first exploring \(I_j(s)\) instead of \(I_k(s)\). Once this proof path is exhausted, \{log\} will attempt the proof by exploring \(I_k(s)\), which will eventually succeed. The net result is \{log\} taking longer than necessary.

Each of the proof obligations in (3) is encoded in \{log\} as the following query:

\[
\text{neg(I_k(S) & a(S,S_) implies I_k(S_))}
\]

where \(_k\), a (S, S_) and S_ correspond to \(I_k, s \xrightarrow{a} s'\) and \(s'\), respectively.
4.2 Security Properties

In order to encode in \( \{ \log \} \) the security properties described in Sect. 2.2.3, we classified them as follows:

A. Properties given in terms of a sequence of operations.
B. Existential properties, i.e. properties establishing the existence of a state where some predicate holds.
C. Universal properties, i.e. classical system properties.

In general, properties in class A. cannot be expressed in \( \{ \log \} \). However, the proof of some of these properties can be divided into two steps: firstly, some predicate is proved to hold for every operation; secondly, the first result is used to prove that any sequence of operations verifies the desired property. The first step is hard, whereas the second is rather simple. \( \{ \log \} \) can be used in the first step by proving that the involved predicate is indeed an invariant (as shown in Sect. 4.1). An example of this class of properties is \texttt{ifOldAppRunThenWasVerified} shown in Fig. 8. As can be seen, this property depends on \( \mathsf{list} \ \mathsf{Action} \), where \( \mathsf{Action} \) is a Coq type containing all the operations of the model. Then, \( \\langle \text{trace } \mathsf{initState} \ \mathsf{l} \rangle \) captures the state trace followed by the system when running from a given initial state. Basically, the property states that if an old application \( \mathsf{isOldApp a } \mathsf{initState} \) can be run in the last state \( \mathsf{canRun a } \mathsf{lastState} \), then it is because it has been verified along the state trace \( \langle \text{verifyOldApp a } \rangle \ \mathsf{l} \). Then, in \( \{ \log \} \) we proved a series of lemmas stating that if we have an old application and an operation of the system (except \texttt{verifyOldApp}) is executed, then that application cannot be run. One such lemma (for the \texttt{install} operation) can be seen in Fig. 9.

A typical property in class B. is \texttt{ExecAutoGrantWithoutIndividualPerms}, shown in Fig. 6. Its \( \{ \log \} \) encoding is the following:
Fig. 9 Installing an application does not allow to execute an old application

as can be seen, the existential quantification of Fig. 6 is implicit in \{log\}. Instead of quantifying over \(p'\) and \(\text{permsA}\) we first get \(\text{PermsA}\) as the image of \(A\) through \(\text{perms}\); if \(A\) is not in \(\text{perms}'s\) domain, \text{applyTo}(\text{perms}, A, \text{permsA})\) fails. Then, we only need \(P_\) to quantify over \(\text{permsA}\) in the \text{exists} constraint. In Coq, \text{Some} \(g\) is part of an \text{option type} which in \{log\} is encoded as \text{grp}(G), where \text{grp} is a functor and \(G\) an existential variable.

As can be seen, the existential quantification of Fig. 6 is implicit in \{log\}. Instead of quantifying over \(p'\) and \(\text{permsA}\) we first get \(\text{PermsA}\) as the image of \(A\) through \(\text{perms}\); if \(A\) is not in \(\text{perms}'s\) domain, \text{applyTo}(\text{perms}, A, \text{permsA})\) fails. Then, we only need \(P_\) to quantify over \(\text{permsA}\) in the \text{exists} constraint. In Coq, \text{Some} \(g\) is part of an \text{option type} which in \{log\} is encoded as \text{grp}(G), where \text{grp} is a functor and \(G\) an existential variable.

When \text{execAutoGrantWithoutIndividualPerms} is executed, \{log\} attempts to find values for the variables satisfying the formula. In this way \{log\} generates a witness (solution) satisfying the query. The following is a simplified, pretty-printed form of that witness.

\[
\textbf{P} = \{V9, \text{grp}(G), \text{dangerous}\}, \\
\textbf{SystemPerm} = \{(P/Q9)\}, \\
\textbf{S} = \{(St,Env)\}, \\
\textbf{St} = \{[\text{apps},(A/Q1)],V1,[\text{grantedPermGroups},(\{A,(G/Q2)\}/Q3)], \\
\{\text{perms},(\{A,\text{permsA}\}/Q4)\},...\}, \\
\textbf{Env} = \{[\text{manifest},(\{A,[V2,V2,V3,(P/Q5),V4,V5]/Q6\}),[\text{cert},(\{A,V6\}/Q7)], \\
\{\text{defPerms},(\{A,V7\}/Q8)\},V8\}
\]

Variables \(V?\) and \(Q?\) are new (existential) variables. This witness is followed by a number of constraints restricting the values the variables at the right-hand side can take. For example, one such constraint is \text{comp}((\{A,A\},Q6,\{\}) meaning that \(A\) is not in the domain of \(Q6\), which is the ‘rest’ of \text{manifest}. This ensures that \text{manifest} is a partial function, as required by \text{validstate}. In order to come out with a more concrete witness every \(Q?\) variable can be replaced by the empty set. For instance, a more concrete value for \text{manifest} can be obtained by replacing \(Q5\) and \(Q6\) with \{\} thus getting \{(\{A,[V2,V2,V3,P,V4,V5]\})\}. \{log\} guarantees that the constraints following a witness are always satisfied by such a simple replacement because those constraints are of an special kind called \text{irreducible constraints} [20, 22–24, 31].

Finally, \text{CannotAutoGrantWithoutGroup}, shown in Fig. 5, is a typical property of class \text{C}. The following is its \{log\} encoding.
cannotAutoGrantWithoutGroup :-
    neg(
        state(S,St) & grantedPermGroups(St,GR) &
        pl(P,dangerous) &
        maybeGrp(P,grp(G)) &
        applyTo(GR,A,GA) &
        G nin GA
    implies neg(grantAuto(SystemPerm,P,A,S,S_))
).

Clearly, the Coq theorem based on proving a universal property becomes a \(\{\text{log}\}\) query attempting to satisfy the negation of that property. The only issue worth noting in this encoding is that exec s (grantAuto p a) s’ ok is implemented by simply calling grantAuto. If the outcome of exec is ok it means that grantAuto could be executed which in \(\{\text{log}\}\) corresponds to grantAuto being satisfiable.

In this case, when cannotAutoGrantWithoutGroup is executed \(\{\text{log}\}\) is unable to find values satisfying the formula so it returns false. If, for some reason, there would have been an error in grantAuto such that cannotAutoGrantWithoutGroup would not hold, then \(\{\text{log}\}\) would return a counterexample when the query is executed. The counterexample would help to find the error.

As a summary, it is reasonable to say that the \(\{\text{log}\}\) code is a straightforward, set-based encoding of the Coq model.

**Remark 4** (An Automatically Verified Prototype) Once all queries representing properties are successfully run, the \(\{\text{log}\}\) program can be regarded as an automatically verified (or certified or correct) prototype of the Android permissions system w.r.t. the proven properties. As we will see in Sect. 4.3, \(\{\text{log}\}\) is able to automatically discharge in a reasonable time 24 out of 27 (≈ 90%) of the properties proposed in the Coq model.

### 4.3 Empirical Evaluation

In this section we provide detailed data on the empirical evaluation resulting after executing all the \(\{\text{log}\}\) queries representing valid state and security properties, described above. The goal of the empirical evaluation is to measure the number of properties that \(\{\text{log}\}\) is able to prove and the amount of time it spends on that. The \(\{\text{log}\}\) code and instructions on how to reproduce this empirical evaluation can be found online.\(^4\)

Table 1 summarizes the results of the empirical evaluation and provides some figures as to compare the \(\{\text{log}\}\) and Coq developments. The upper part of the table gives some figures about the size of both projects. The Coq development encompasses the model described in Sect. 2.2 and a Coq functional implementation proved to refine the model. In this way we are comparing all the work needed to get an executable prototype in Coq as well as in \(\{\text{log}\}\). As can be seen, the \(\{\text{log}\}\) code is larger than the Coq code in terms of LOC but the latter needs forty times more characters than the former (recall granPerm). \(\{\text{log}\}\) lines tend to be really shorter—many times only one constraint per line is written. Concerning proofs, Coq needs almost twice as many LOC than \(\{\text{log}\}\) although in terms of characters Coq uses not many more than \(\{\text{log}\}\). Here, however, it is important to observe that most of the proof text in \(\{\text{log}\}\) can be automatically generated. In effect, 90% of the proof text in \(\{\text{log}\}\) corresponds to the statement of the invariance lemmas that ensure that each operation preserves valid

\(^4\) [https://www.clpset.unipr.it/SETLOG/APPLICATIONS/android.zip](https://www.clpset.unipr.it/SETLOG/APPLICATIONS/android.zip).
Table 1  Summary of the empirical evaluation

| Project Size (specification + proof) | LOC (K) | Char (K) |
|------------------------------------|---------|----------|
| Model Proofs                       |         |          |
| Model Proofs                       |         |          |
| $(\log)$                           | 4.2     | 11       |
| Coq                                | 3.5     | 18       |

| Performance of Automated Proof $(\log)$ |
|-----------------------------------------|
| COQ                                    |
| Lemmas Queries                          |
| Time (s)                                |
| Valid-state invariance lemmas            | 13      | 13       | 756 | 920         |
| Security properties                     | 14      | 11       | 45  | 381         |
| TOTALS                                  | 27      | 24       | 801 | 1,302       |

states, and not to the actual proofs. These statements can be automatically generated by conveniently annotating clauses representing operations and invariants and then calling a verification condition generator which generates lemmas such as 3. On the other hand, most of the proof text in Coq corresponds to the proof commands written by the user. However, these proofs use standard Coq strategies and tactics that do not apply automated methods such as those proposed by Chlipala [15] for engineering large scale formal developments. The use of such methods could reduce the number of lines and the number of tactics used in the proofs.

In turn, the lower part of the table provides figures about the performance of $(\log)$ concerning automated proof. As we have said in Sect. 2.2, there are 13 valid-state invariance lemmas and 14 security properties defined and proved in Coq. $(\log)$ is able to automatically discharge all the invariance lemmas and 11 of the security properties. That is 24 out of 27 ($\approx 90\%$) Coq proofs are automatic in $(\log)$. In order to prove the 13 invariance lemmas we defined 756 $(\log)$ queries. Recall from Sect. 4.1 that some Coq predicates defining the notion of valid state are split in two or more $(\log)$ clauses (e.g. notDupPerm). $(\log)$ needs 920 s ($\approx 15$ m) in order to execute all these queries. Same considerations apply to security properties. That is, the 11 properties are encoded as 45 $(\log)$ queries which are discharged in 381 s ($\approx 6$ m). For example, some security properties of class A. (Sect. 4.2) are expressed as several state invariants. In summary, $(\log)$ needs 1,302 s ($\approx 22$ m) to prove 24 properties expressed as 801 satisfiability queries.

The above figures were obtained on a Latitude E7470 (06DC) with a 4 core Intel(R) Core™ i7-6600U CPU at 2.60GHz with 8 Gb of main memory, running Linux Ubuntu 18.04.6 (LTS) 64-bit with kernel 4.15.0-184-generic. $(\log)$ 4.9.8-11 h over SWI-Prolog (multi-threaded, 64 bits, version 7.6.4) was used during the experiments.

5 Discussion and Related Work

Besides the Coq model discussed in Sect. 2.2, we have developed a Coq functional implementation [30]. Then we demonstrated, using Coq, that this implementation satisfies the model. Finally, thanks to the program extraction mechanism offered by Coq, we obtained a Haskell implementation of the Android permission system satisfying the model. This is the way a
verified prototype is usually generated in Coq. Conversely, in \{log\} it is enough to write the specification and then to prove properties to gain confidence on its correctness.

As the empirical evidence shows, combining the automated proof capabilities of \{log\} with the proving power of Coq would be beneficial in terms of human effort and development time, if the tools are integrated in some way. \{log\} is able to automatically discharge all but three of the proposed properties, thus saving a lot of time and human effort; these three properties are manually proved in Coq. Hence, the question is which is the better integration strategy to make \{log\} and Coq to work together. Traditionally, automated theorem provers (ATP) have been integrated into interactive theorem provers (ITP) such as Coq. For example, CoqHammer \[29\] uses external ATPs to automate Coq proofs. Likewise, Isabelle/HOL uses Sledgehammer extended with SMT solvers \[10\]. In these cases, the current goal in the ITP is encoded in the input languages of the external solvers.

We instead propose to integrate \{log\} with Coq the other way around: encoding \{log\} models as Coq specifications.\(^5\) That is, we envision the following methodology:

1. Write a \{log\} model.
2. Prove as many properties as possible using \{log\}.
3. If there are unproven properties, encode the \{log\} model as a Coq specification.
4. Use Coq to prove the remaining properties.

In this way, the \{log\} model becomes a correct-by-construction prototype. This implies that it would not be necessary to write a Coq abstract model, its implementation and perform the refinement proofs in order to get a certified executable prototype.

The justification for encoding \{log\} models as Coq specifications can be sought in Figs. 4 and 7. As we have shown, grantPerm is encoded in \{log\} as line 11 of Fig. 7. Consider for a moment (automatically) translating grantPerm into \{log\} in a way that line 11 is obtained. The task seems overwhelming. It might be tempting to consider more literal Coq-to-\{log\} translations where the \{log\} code would look much as grantPerm. The problem with this approach is that it will severely reduce the automated proving capabilities of \{log\}. Indeed, \{log\} is able to discharge many proofs because predicates like grantPerm are encoded as simple \{log\} constraints. That is, \{log\} “knows” what to do with foplus but it will not necessarily know what to do if that is written in a different way. On the other hand, (automatically) encoding \{log\} constraints (e.g. foplus) in Coq is rather easy. Considering the definition of foplus in equation (2), it is easy to see that it can be encoded in terms of set equality and relational composition which are already defined in Coq. If duly unfolded, the resulting translation would look much as grantPerm. Evidently, Coq is more expressive than \{log\}, making the proposed translation easier than the opposite. Once the \{log\} model is encoded as a Coq specification, properties can be proved as usual.

We believe that one of the core constructs that allow \{log\} to discharge many proof obligations are RUQ. Other tools (e.g., SMT solvers) provide (unrestricted) universal quantification. Although universal quantification is important to gain expressiveness it is harder to deal with when performing automated proofs. Given that RUQ quantify over a finite set, a set solver can deal with them in a more controlled and decidable way \[25, 27\]. At the same time, RUQ provide just the necessary expressiveness when it comes to software specification.

Sooner or later, automated proof hits a computational complexity wall that makes progress extremely difficult. We use hypothesis minimization and predicate delay to move that wall as far away as possible. Hypothesis minimization consists in calling \{log\} to prove a property just with the necessary hypothesis. That is, instead of calling \{log\} to prove \(p \land q \implies r\)

\(^5\) In this work the Coq model was encoded in \{log\} simply because our collaboration started when the Coq model was already available.
we call it to prove \( q \implies r \) if \( p \) does not contribute to the proof. This is particularly useful when proving invariance lemmas. When proving that operation \( t \) preserves invariant \( p \), our first attempt is to prove \( p(s) \land t(s, s') \implies p(s') \). If this fails, we use the counterexample to find out what other invariants must be added as hypothesis. Then, we attempt \( q(s) \land p(s) \land t(s, s') \implies p(s') \), for some other invariant \( q \). Although this approach requires some manual work, it is much less and less complex than an interactive proof. In fact, most of the invariance lemmas are discharged in the first attempt.

Predicate delay instructs \{log\} to delay the processing of a given predicate until nothing else can be done. This is achieved by means of \{log\}'s delay directive. Similar techniques are used, for instance, by SMT solvers. For example Dafny [38], provides the opaque attribute and the reveal directive to selectively hide irrelevant predicates from Z3. Hawblitzel et al. [34, Section 6.3.2] use these constructs and add to Dafny more such constructs in order to tame the complexity of the correctness proofs of distributed systems.

Sometimes these techniques are not enough to make the solver to finish in a reasonable time. As we have pointed out, \{log\} is unable to prove three properties that were proved in Coq. These proofs require less than 500 LOC of Coq code meaning a reduced manual effort (\( \approx 2\% \) of the total Coq proof effort). This is another piece of evidence suggesting that the proposed integration between \{log\} and Coq could work in practice.

\{log\} has been used in a similar fashion with other systems. It can discharge in 2 s all the 60 proof obligations proving the correctness of the Bell-LaPadula security model [4, 5] w.r.t. to the security condition and the *-property [25]. The Tokeneer ID Station (TIS) was proposed by the NSA as a benchmark for the production of secure software. NSA asked Altran UK to provide an implementation of TIS conforming to Common Criteria EAL5 [17]. Altran UK applied its own Correctness by Construction development process to the TIS software including a Z specification of the user requirements. Cristiá and Rossi [26] encoded that Z specification in \{log\} as well as all the proof obligations set forth by the Altran UK team as 523 proof queries. \{log\} is able to prove all the 523 queries in 14 min. Boniol and Wiels proposed a real-life, industrial-strength case study, known as the Landing Gear System (LGS) [11]. Mammar and Laleau [41] developed an Event-B specification of the LGS on the Rodin platform [16]. This specification was encoded in \{log\} along with all the proof obligations generated by Rodin [21]. The tool discharges all the 465 proof obligations in less than 5 min.

Few works study the permission system of Android by using a formal specification language. An example of this is the work of Shin et. al. [49], where they specified an abstract model of the Android permission system very similarly to the way we did in our last work. The main difference, though, is that the work by Shin is based on an older version of Android and was never updated. More examples of Coq being used to study Android at a formal level are the recent work carried out by El-Zawawy et. al. [32] or the CrashSafe tool [37]. These works are focused, though, on studying Inter-Component Communication (ICC) properties rather than security properties of the permission system.

On the other hand, not everyone chooses Coq to formalize and analyze a model of Android. For instance, Bagheri et al. [3] formalized and studied the Android permission system using Alloy [36]. As another example of a formal language being used to study Android, we can mention the Terminator tool [46] which uses the TLA+ model checker to analyze and identify permission induced threats. The main difference between these approaches and a Coq formalization, is that the latter provides stronger guarantees about the safety and security properties of the platform although requires more human effort to identify potential flaws. Our goal in this work is to bring together the robustness of the properties that one can get from a Coq model combined with the automation that tools like Alloy or \{log\} can provide. However, it should be observed that Alloy can only prove that formulas are satisfiable, whereas \{log\}
can prove both, satisfiability and unsatisfiability. There are more works using formal methods to study a specific aspect of Android which do not necessarily conduct formal verification [33, 42, 51]. These approaches require a high amount of human effort, not only for identifying flaws but also to prove any desired property.

6 Conclusions

We present a \{log\} model from one in Coq that constitutes an automatically verified executable prototype of the Android permissions system. In particular, we show in detail how the Coq model is encoded in the \{log\} language and how automated proofs are performed. We conclude that \{log\} can be used as an effective automated prover for systems such as the Android permissions system. In this way much of the manual, expert work needed to prove properties in Coq can be avoided.

Detailed data on the empirical evaluation resulting after executing all the proofs in \{log\} is provided. The empirical evidence shows that the combination of the automated proof capabilities of \{log\} with the proving power of Coq would be beneficial in terms of human effort and development time.

We have not seen other verification efforts combining Coq and \{log\} to produce a certified prototype of a critical system.

Finally, we discuss possible ways to integrate Coq and \{log\} as to provide a framework featuring automated proof and prototype generation. Advancing in this direction for the analysis of critical systems (in general) is part of our future work.

Funding No funding.

Declarations

Conflict of interest No conflicts of interest.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Android Developers: Application Fundamentals. http://developer.android.com/guide/components/fundamentals.html. Accessed Jan 2023
3. Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the android permission system. Formal Aspects Comput. 30(5), 525–544 (2018). https://doi.org/10.1007/s00165-017-0445-z
4. Bell, D.E., LaPadula, L.: Secure computer systems: mathematical foundations. MTR 2547, The MITRE Corporation (May 1973)
5. Bell, D.E., LaPadula, L.: Secure computer systems: mathematical model. ESD-TR 73-278, The MITRE Corporation (November 1973)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development—Coq’Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2004). https://doi.org/10.1007/978-3-662-07964-5
7. Betarte, G., Campo, J.D., Luna, C.D., Romano, A.: Verifying Android’s permission model. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical aspects of computing—ICTAC 2015—12th international colloquium Cali, Colombia, October 29–31, 2015. In: Proceedings. Lecture Notes in Computer Science, vol. 9399, pp. 485–504. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-25150-9_28
8. Betarte, G., Campo, J.D., Luna, C., Romano, A.: Formal analysis of Android’s permission-based security model. Sci. Ann. Comput. Sci. 26(1), 27–68 (2016). https://doi.org/10.7561/SACS.2016.1.27
9. Betarte, G., Campo, J.D., Gorostiaga, F., Luna, C.: A certified reference validation mechanism for the permission model of Android. In: Fioravanti, F., Gallagher, J.P. (eds.) Logic-Based Program Synthesis and Transformation—27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10855, pp. 271–288. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-94460-9_16
10. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. J. Autom. Reason. 51(1), 109–128 (2013). https://doi.org/10.1007/s10817-013-9278-5
11. Bonioli, F., Wiels, V.: The landing gear system case study. In: Bonioli, F., Wiels, V., Ameur, Y.A., Schewe, K. (eds.) ABZ 2014: The Landing Gear Case Study—Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2–6, 2014. Proceedings. Communications in Computer and Information Science, vol. 433, pp. 1–18. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-07512-9_1
12. Cantone, D., Ferro, A., Omodeo, E.: Computable Set Theory. Clarendon Press, Oxford (1989)
13. Cantone, D., Omodeo, E.G., Policriti, A.: Set theory for computing - from decision procedures to declarative programming with sets. In: Monographs in Computer Science. Springer, Berlin (2001). https://doi.org/10.1007/978-1-4757-3452-2
14. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.A.: Analyzing inter-application communication in android. In: Agrawala, A.K., Corner, M.D., Wetherall, D. (eds.) Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011, pp. 239–252. ACM (2011). https://doi.org/10.1145/1999995.2000018
15. Chlipala, A.: Certified Programming with Dependent Types—A Pragmatic Introduction to the Coq Proof Assistant. MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types
16. Coleman, J., Jones, C., Oliver, I., Romanovsky, A., E.Troubitsyna: RODIN (Rigorous open Development Environment for Complex Systems). In: Fifth European Dependable Computing Conference: EDCC-5 Supplementary Volume, pp. 23–26 (2005)
17. Common Criteria Recognition Arrangement: common criteria for information technology security evaluation, part 1: introduction and general model, version 3.1. release 5. Technical report (2017). https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf
18. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. 76(2/3), 95–120 (1988). https://doi.org/10.1016/0890-5401(88)90005-3
19. Cristiá, M., Rossi, G.: A set solver for finite set relation algebra. In: Desharnais, J., Guttmann, W., Joosten, S. (eds.) Relational and Algebraic Methods in Computer Science—17th International Conference, RAMICS 2018, Groningen, The Netherlands, October 29–November 1, 2018. Proceedings of Lecture Notes in Computer Science, vol. 11194, pp. 333–349. Springer, Berlin (2018). https://doi.org/10.1007/978-3-030-02149-8_20
20. Cristiá, M., Rossi, G.: Solving quantifier-free first-order constraints over finite sets and binary relations. J. Autom. Reason. 64(2), 295–330 (2020). https://doi.org/10.1007/s10817-019-09520-4
21. Cristiá, M., Rossi, G.: An automatically verified prototype of a landing gear system (2021). CoRR arXiv:2112.15147
22. Cristiá, M., Rossi, G.: A decision procedure for a theory of finite sets with finite integer intervals. CoRR abs/2105.03005 (2021)
23. Cristiá, M., Rossi, G.: Integrating cardinality constraints into constraint logic programming with sets. Theory Pract Logic Program 1–33 (2021)
24. Cristiá, M., Rossi, G.: Automated reasoning with restricted intensional sets. J. Autom. Reason. 65(6), 809–890 (2021). https://doi.org/10.1007/s10817-021-09589-w
25. Cristiá, M., Rossi, G.: Automated proof of Bell-LaPadula security properties. J. Autom. Reason. 65(4), 463–478 (2021). https://doi.org/10.1007/s10817-020-09577-6
26. Cristiá, M., Rossi, G.: An automatically verified prototype of the Tokeneer ID station specification. J. Autom. Reason. 65(8), 1125–1151 (2021). https://doi.org/10.1007/s10817-021-09602-2
27. Cristiá, M., Rossi, G.: A set-theoretic decision procedure for quantifier-free, decidable languages extended with restricted quantifiers. CoRR abs/2208.03518 (2022). https://doi.org/10.48550/arXiv.2208.03518. Under consideration in Journal of Automated Reasoning
28. Cristiá, M., Rossi, G., Fredman, C.S.: {log} as a test case generator for the Test Template Framework. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM. Lecture Notes in Computer Science, vol. 8137, pp. 229–243. Springer, Berlin (2013)
29. Czajka, L., Kaliszyn, C.: Hammer for Coq: automation for dependent type theory. J. Autom. Reason. 61(1–4), 423–453 (2018). https://doi.org/10.1007/s10817-018-9458-4
30. De Luca, G., Luna, C.: Towards a certified reference monitor of the Android 10 permission system. In: de’Liguoro, U., Berardi, S., Altenkirch, T. (eds.) 26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2–5, 2020, University of Turin, Italy. LIPIcs, vol. 188, pp. 3–1318. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.TYPES.2020.3

31. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. 22(5), 861–931 (2000)

32. El-Zawawy, M.A., Faruki, P., Conti, M.: Formal model for inter-component communication and its security in android. Computing 104(8), 1839–1865 (2022). https://doi.org/10.1007/s00607-022-01069-2

33. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing android’s permission system. In: Foresti, S., Yung, M., Martinelli, F. (eds.) Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10–12, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7459, pp. 1–18. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-33167-1_1

34. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015, pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428

35. International Data Corporation: Smartphone market share. Technical report, International Data Corporation (IDC) (2022)

36. Jackson, D.: Alloy: A logical modelling language. In: Bert, D., Bowen, J.P., King, S., Waldén, M.A. (eds.) ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2651, p. 1. Springer, Berlin (2003). https://doi.org/10.1007/3-540-44880-2_1

37. Khan, W., Ullah, H., Ahmad, A., Sultan, K., Alzahrani, A.J., Khan, S.D., Alhumaid, M., Abdulaziz, S.: Crashsafe: a formal model for proving crash-safety of android applications. Hum. Centric Comput. Inf. Sci. 8, 21 (2018). https://doi.org/10.1186/s13673-018-0144-7

38. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning—16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer, (2010). https://doi.org/10.1007/978-3-642-17511-4_20. https://doi.org/10.1007/978-3-642-17511-4_20

39. Letouzey, P.: Programmation fonctionnelle certifiée – l’extraction de programmes dans l’assistant Coq. PhD thesis, Université Paris-Sud (July 2004)

40. Luca, G.D., Luna, C.: Formal verification of the security model of Android 10: Coq code. https://github.com/g-deluca/android-coq-model. Accessed Jan 2023

41. Mammar, A., Laleau, R.: Modeling a landing gear system in Event-B. Int. J. Softw. Tools Technol. Transf. 19(2), 167–186 (2017). https://doi.org/10.1007/s10009-015-0391-0

42. Micinski, K.K., Fetzer-Degges, J., Jeon, J., Foster, J.S., Clarkson, M.R.: Checking interaction-based declassification policies for android using symbolic execution. In: Pernul, G., Ryan, P.Y.A., Weippl, E.R. (eds.) Computer Security—ESORICS 2015—20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9327, pp. 520–538. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-24177-7_26

43. Open Handset Alliance: Android Project. https://source.android.com. Accessed Jan 2023

44. Paulin-Mohring, C.: Inductive definitions in the system coq - rules and properties. In: Bezem, M., Groote, J.F. (eds.) Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA ’93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings. Lecture Notes in Computer Science, vol. 664, pp. 328–345. Springer, Berlin (1993). https://doi.org/10.1007/BFb0037116

45. Rossi, G.: {log}. Accessed 2022

46. Sadeghi, A., Jabbarvand, R., Ghorbani, N., Bagheri, H., Malek, S.: A temporal permission analysis and enforcement framework for android. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27–June 03, 2018, pp. 846–857. ACM (2018). https://doi.org/10.1145/3180155.3180172

47. Sbirlea, D., Burke, M.G., Guarnieri, S., Pistoia, M., Sarkar, V.: Automatic detection of inter-application permission leaks in Android applications. IBM J. Res. Dev. 57(6). https://doi.org/10.1147/JRD.2013.2284403

48. Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E.: Programming with sets—an introduction to SETL. In: Texts and Monographs in Computer Science. Springer, New York (1986). https://doi.org/10.1007/978-1-4613-9575-1
49. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the Android framework. In: Elmagarmid, A.K., Agrawal, D. (eds.) Proceedings of the 2010 IEEE Second International Conference on Social Computing, SocialCom/IEEE International Conference on Privacy, Security, Risk and Trust, PASSAT 2010, Minneapolis, Minnesota, USA, August 20-22, 2010, pp. 944–951. IEEE Computer Society (2010). https://doi.org/10.1109/SocialCom.2010.140
50. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall International (UK) Ltd., Hertfordshire (1992)
51. Talegaon, S., Krishnan, R.: A formal specification of access control in android with URI permissions. Inf. Syst. Front. 23(4), 849–866 (2021). https://doi.org/10.1007/s10796-020-10066-9
52. The Coq Team: The Coq Proof Assistant Reference Manual—Version V8.12.0. LogiCal Project, Palaiseau, France (2020). LogiCal Project. http://coq.inria.fr

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

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.