Android Platform Modeling and Android App Verification in the ACL2 Theorem Prover

Eric Smith and Alessandro Coglio

Kestrel Institute, Palo Alto, U.S.A.
http://www.kestrel.edu
{eric.smith,coglio}@kestrel.edu

Abstract. We present our work in using the ACL2 theorem prover to formally model the Android platform and to formally verify Android apps. Our approach allows the verification of the full functional correctness of apps as well as security properties. It also lets us detect or prove the absence of "functional malware", malicious app functionality that is triggered by complex conditions on state and that causes the app to calculate the wrong results or otherwise behave incorrectly. Our formal Android model is an executable simulator of a growing subset of the Android platform, and app proofs are done by automated symbolic execution of the app's event handlers using the formal model. By induction, we prove that an app satisfies an invariant, including the correctness properties of interest, for all possible sequences of events.

1 Introduction

Android devices [32] are vulnerable to security compromises carried out by rogue apps that may abuse the user's trust by masquerading as benign apps [12,40]. The Android security mechanisms are coarse and complex [11,34] and may be bypassed via exploitable flaws in the platform [2,24].

A more detailed characterization of an app's behavior, especially its access to user data, can enable users to make more informed decisions about trusting and installing the app. A suitable formal specification of the app can be used for this purpose, and trust can be established via a formal proof that the app's code satisfies the specification. This requires a formal model of the platform that the app runs on—both language and API.

The work described in this paper contributes to the goal of establishing trust in apps based on formal specifications and proofs. We used the ACL2 theorem prover [39] to build a formal model of a subset of the Android platform that supports non-trivial apps. We developed a proof methodology based on induction and symbolic execution of the app's event handlers, showing that each handler preserves the app's invariant, which includes all properties of interest, including functional correctness.

We applied this proof methodology to verify the full functional correctness of a slightly simplified version of a calculator app written by others. For a version of the app that contains malware, the correctness proof fails in a way that reveals the malware. In the process of verifying the app, we also uncovered a subtle functional bug that may be representative of malware that is triggered by complex conditions on an app's state and whose malicious action is the calculation of incorrect results. This "functional malware" differs from more explicit, and potentially more easily detectable, malware that, for example, sends private user data to a remote server when the device is in a certain location at a certain time. The latter kind of malware makes API calls to test the trigger conditions and perform the malicious actions, while functional malware may not make any suspicious API calls. For example, functional malware in a navigation app could deliberately lead users off course, perhaps even directing them to dangerous places.

Our approach is sound, precise, and high-assurance, in contrast to existing approaches for vetting Android apps. Static analysis is imprecise, leading to false warnings, sometimes unsound, and cannot check arbitrary functional correctness properties. Dynamic analysis cannot cover all possible cases. Manual code inspection is not high-assurance, because hidden malicious functionality can be overlooked or misunderstood. Our approach can prove virtually any true property about an app, with high assurance. Its main disadvantage is that it requires significant user effort, but we are working to improve the automation of the proof process.

Our work makes the following contributions:

- A formal model of a non-trivial subset of the Android platform.
- A formal proof methodology for Android apps.

2 Background

2.1 Android

Most Android apps are written in Java [31]. Besides using a subset of the standard Java API, these apps use the Android API, which provides access to hardware devices (camera, GPS, etc.), GUI elements (buttons, text boxes, etc.), inter-app communication (e.g., to open a given URL in a web browsing app), and so on. In addition to the Java source files, an app contains other resources, which often take the form of XML files. An app's Java source code is compiled to Java Virtual Machine (JVM) bytecode [23] using a standard Java compiler. The Android development tools are used to convert the JVM bytecode to Dalvik bytecode [32], which is assembled with the XML and other resource files (e.g., images) into an installable app package.

An Android app is structured in terms of 'activities', each of which is a single "screen" in the app's GUI. Within an activity are various 'views'—rectangular regions of the screen that represent GUI elements, such as text boxes and buttons that can be clicked. Events in Android include click events for these views.

An app can register listeners for such events, either statically in its layout XML or programmatically by calling `setOnClickListener()`. When these events occur, the Android GUI thread invokes the appropriate methods of the registered listeners. An app's XML 'manifest' indicates, among other things, the initial activity to be created when the app starts.

Android also includes lifecycle events (*Create*, *Start*, *Resume*, *Restart*, *Pause*, *Stop*, and *Destroy*) that can be dispatched to the app. The sequencing of these events must be consistent with the activity lifecycle state machine [31] (a typical flow is: *Create*, *Start*, *Resume*, *Pause*, *Stop*, *Destroy*) but can otherwise occur at any time. For example, a *Pause* event may occur when another app opens in front of the current app. Apps typically implement handlers to respond to these events (e.g., to save data when the app is paused) by overriding methods of the `Activity` class, such as `onPause()`.

Various entities belonging to the app are identified using numeric resource IDs. These resource IDs are defined in special classes, namely the `R` class ('resource' class) and its inner classes, generated by the Android development tools. For example, an XML layout entity `<Button android:id="@+id/btnSeven" ...>` will cause the `R$id` class to contain a final static field called `btnSeven` whose `ConstantValue` attribute is some large, unpredictable number, e.g., 2131034114. In Java source code, the button object can be obtained by the method call `findViewById(R.id.btnSeven)`, but in the bytecode only the numeric ID is present.

Android includes a permission mechanism to limit apps' access to hardware and other resources. For example, an app must possess the `INTERNET` permission to open network sockets and the `CALL_PHONE` permission to initiate phone calls. An app declares, in its XML manifest, the set of permissions that it requests. When an app is about to be installed, the requested permissions are shown to the user, who decides whether to proceed with the installation, and thus grant the app all the requested permissions. This permission mechanism is coarse-grained: for instance, the `INTERNET` permission gives an app carte blanche to connect to any host at any time to send any data.

Malware Several kinds of malware affect Android devices [12,40]. Tools like [16] can be effective at detecting malware that exfiltrates private user data by (necessarily) making suspicious API calls. The mere presence of certain API calls may be suspicious, e.g., an app that opens a network connection, when the app's purported functionality does not involve the network. The presence of an API call may be legitimate, but the information that flows to the API calls may be suspicious, e.g., an app reads a user's contacts and sends them over the network, when the app's purported functionality does not include that.

A more stealthy kind of "functional malware" may not exfiltrate private user data, and instead intentionally calculate incorrect results. The severity of this kind of malware depends on how much the user relies on the app calculating correct results: it may range from an annoyance to loss of life, e.g., if a military navigation app sends a squad off-course to a dangerous place. Functional malware

may be triggered under complex conditions on an app's state variables, eluding detection via code inspection. Functional malware may involve API calls, but not necessarily suspicious ones; or it may not involve any API calls.

Unlike many other approaches, our work addresses functional malware. Of course, it also addresses inadvertent errors. The difference between functional malware and an unintentional bug is one of developer's intent; but the impact may be similar. Our app verification approach establishes functional correctness, ruling out both intentional and unintentional bugs.

2.2 ACL2

The ACL2 theorem prover [39] consists of a first-order specification language based on side-effect-free Common Lisp and automated proof methods for reasoning about programs and models written in the language. Two strengths of ACL2 are its sophisticated term rewriter and its heuristic application of induction [4]. ACL2 supports reasoning about programs written in languages other than its native Common Lisp dialect via embeddings that capture the languages' semantics in terms of ACL2's native language. Below we describe how we use this approach to reason about JVM bytecode representing Android apps.

3 Platform Modeling

Since our motivation for modeling the Android platform is app verification, our formal model describes not the internal structure and layers of the platform stack, but the top-level interface that the platform provides to apps. This interface consists of the language that apps are written in and the API calls exchanged between apps and platform, including callbacks.

3.1 Formal JVM Bytecode Model

To reason about an Android app, we intercept its JVM bytecode during compilation (before Dalvik bytecode is generated). To assign semantics to this bytecode, we defined in ACL2 a formal model that is an executable interpreter of the Java Virtual Machine [38]. Our model is similar to the M5 model developed by J Moore and others [29], but covers more features (e.g., exceptions, string interning, and class initialization). Theorems about JVM bytecode programs are expressed using this formal model; we prove that when the program of interest is executed on the model, starting from a state where certain properties hold, then certain other properties always hold on the resulting state. This follows the style pioneered in [27].

While we do not consider our JVM model to be a novel contribution of this paper, we summarize its behavior here for concreteness. The state of the JVM in our model includes the Java heap, static area (where static fields are stored), and, for each thread, a call stack that includes invocation frames for each method that

the thread is currently executing. Also included are auxiliary data structures for synchronization and locking, string interning, etc.

Each JVM instruction is modeled by specifying the effect on the JVM state when that instruction is executed. For example, the `iadd` instruction for integer addition is modeled as follows:

```
(defun execute-IADD (th s)
  (modify th s
          :pc (+ 1 (pc (top-frame th s)))
          :stack (push (bvplus 32
                               (top (pop (stack (top-frame th s))))
                               (top (stack (top-frame th s))))
                       (pop (pop (stack (top-frame th s)))))))
```

The function `execute-IADD` modifies the data structures of thread `th` in the JVM state `s`. In particular, it pops two operands off of the operand stack in the top invocation frame of the call stack, adds them, and pushes the sum back onto the operand stack. It then increments the program counter `:pc` by 1, which is the length of the `iadd` instruction.

To run an entire program, we repeatedly step the machine state by fetching and dispatching on the next instruction. We use ACL2's `defpun` utility to soundly introduce the JVM interpreter as a partial function [25].

A crucial feature of our JVM model is that, in addition to running bytecode programs on concrete inputs, it can be used for symbolic execution of bytecode programs on arbitrary inputs. A typical theorem says, in essence, "When we run the JVM model on this bytecode program, for any input satisfying this predicate, the resulting state has the following properties." The symbolic execution is performed using the ACL2 rewriter to repeatedly step and simplify the state, symbolically executing one instruction at a time and building up a symbolic representation of the current state in terms of the symbolic inputs. This technique is standard in the ACL2 community. In this way, our formal JVM model captures the semantics of the JVM bytecode language and allows us to reason about the code that constitutes Android apps.

3.2 Formal Android Model

We extended the formal JVM model described above to a formal model of the Android platform, capable of executing and reasoning about simple Android apps. A state in the Android model contains a JVM state and several additional Android-specific state components More precisely, our model of the Android state contains:

- A JVM state, as discussed above. This contains the persistent data used by the app, including its heap and static fields.
- The app's activity stack, including the current activity on top of the stack, and any activities that are currently paused, below the top activity.
- The set of currently allowed events (e.g., button clicks) for which the app has registered event handlers.

- A parsed representation of the app's manifest—see Sect. 2.
- The app's layout information, parsed from the app's XML layout files and indexed by the layouts' numeric IDs. This includes information about the views (e.g., buttons) in the app's GUI and their associated event handlers (e.g., `onClick` listeners) and is used by our model of the `setContentView()` API method when it constructs the GUI for an activity.
- A map from the addresses of `View` objects to their listeners, used to dispatch control when handling events. A listener is a pair of a method (often, but not always, the `onClick()` method of some class) and an object on which to invoke the method (often this is an `Activity` object or an instance of an anonymous class whose sole purpose is to define the listener). This map is updated by our model of the `setOnClickListener()` API method.
- A map from symbolic string names of views, used in the layout XML, to the corresponding numeric resource IDs. This is used to translate events from user-meaningful form to internal form. We build this map by inspecting the names and values of the static fields of the `R$Id` resource class generated when the app is built.
- A map from resource IDs to the addresses of their corresponding `View` objects. This is used to determine the actual objects on which to dispatch events (e.g., click events) and by our model of the `findViewById()` API method.
- The API call history, a ghost variable that lets us reason about the API calls that the app has (and, critically, has not) made, including a record of the event whose handler made each API call.
- The event history, a ghost variable that lets us talk about the sequence of events given to the app so far. If we are verifying that the app implements an abstract state machine, we can abstract this event history and feed it to the abstract state machine. The resulting abstract state should then be the abstraction of the machine's current concrete state. Proving that this property is preserved by all event handlers in the app is the core step of our app proof methodology described below.
- The event currently being handled, if any, so that we can record in the API history which event was being handled when the API call was made. API calls may be allowed for some events but not others. For example, a sound recorder app may be allowed to start recording only when the user presses the *Record* button.

Event Handling Our Android model supports running an app on a sequence of input events, by executing their event handlers in order. This can be done on a concrete sequence of events, to test an app. More importantly, it can be used for proof. We prove that, for any sequence of events, running the app's handlers for those events preserves the app's invariant. At this level, events are represented in terms that are meaningful to the user. For example, `(:resume)` represents the event that resumes the current activity, and `(:click "myButton")` represents a click of the button whose name in the layout is `myButton`. In order to actually

handle these events, our model must determine the objects on which the handler methods should be invoked, so it first converts the events into an internal form. For lifecycle events, this adds to the event the heap address of the topmost activity object on the activity stack, giving something like `(:resume 12345)`. Click events are internalized by mapping the symbolic name of the button to a numeric resource ID and then to the actual address of the `View` object with that ID, giving something like `(:click 6789)`. Currently our model only handles lifecycle events and click events, but adding support for other events should be straightforward.

Once the event has been elaborated to internal form, we dispatch it to the appropriate handler by executing the code for the handler using the underlying JVM model. For a lifecycle event, we execute an `invokevirtual` instruction for the appropriate handler method (e.g., `onResume()`) on the given `Activity` object, which causes the app's `onResume()` handler method to run. Such methods almost always begin by calling through to the corresponding method of the parent class, e.g., `super.onResume()`. This causes code from the Android API implementation to run, e.g., `android.app.Activity.onResume()`. Our model includes special modeling for these lifecycle API calls. For example, the model for `onResume()` causes the `onClick` listeners in the resuming activity to again be added to the set of allowed events. To handle a click event, assuming it is already in internal form, we look up the `onClick` listener for the given `View` object and call the indicated method. In our model, handlers execute to completion and cannot be interrupted. This corresponds to Android's use of an app's main 'UI thread' to execute its handlers. Future work would include adding support for background services, which an app can use to offload expensive computation from its UI thread.

The sequential processing of events in our model corresponds to the way in which the Android platform internally enqueues events and delivers them to an app's unique UI thread. By proving properties over all possible event sequences, we ensure that the properties hold no matter how the Android platform enqueues and delivers the events.

Events that are not currently allowed by the app (according to the set of allowed events in the Android state) are ignored, e.g., a click on a view that has no registered `onClick` listeners, or an illegal lifecycle event, such as stopping an activity that has not been started. Every event is also recorded in the event history, so that the invariant can refer to the state that the app should be in, given the events seen so far.

3.3 Formal API Model

A major challenge in reasoning about Android apps is to properly model calls to API methods. We are following a "demand-driven" approach in which we add models of API methods as we encounter calls to them in apps that we want to verify. Some methods such as `sendTextMessage()` do not really need to be modeled because they affect only the external world, not the state of the app itself: we simply record them in the API history, so that we can express properties

such as "the app has not sent any text messages", and continue with execution. When the API call does affect the app's state, if possible we simply execute on our model the actual code of the API from the Android implementation. API calls treated this way include many calls in `java.lang` (e.g., dealing with `String`s and `Enum`s) and setters and getters such as `Activity.setTitle()` and `View.isClickable()`. There are situations where simply executing the API call does not work, either because the code is unavailable (e.g., native methods) or too complicated, or because it affects parts of the Android state that we model. To model such methods, we define executable ACL2 functions and include them in our Android model. Methods that are modeled in this way include `setOnClickListener()`, `findViewById()`, `setContentView()`, and the activity lifecycle event handlers `onStart()`, `onResume()`, etc.

Our model of running an app begins by building an initial Android state for the app (where many components, such as the API history, are initially empty) and then calling the app's `onCreate()` method. Further events are then handled in order.

4 App Verification

Our platform model provides a formal semantics for non-trivial Android apps. This allows us to formally prove that apps satisfy their functional specifications, which implies the absence of the kind of functional malware discussed in Sect. 1.

Our methodology is based on formulating an invariant for the app: a predicate over states of the Android model that is preserved as the app runs. The invariant characterizes correct behavior, often using an abstraction to a high-level state machine, and also makes many Android-specific assertions, such as specifying the set of currently active event listeners. Each event is proved to preserve the invariant, using the ACL2 rewriter to perform symbolic execution, as described below. Failed proofs may require the invariant to be strengthened. Once an inductively-strong invariant is obtained, an induction over event sequences establishes that the invariant holds for all possible event sequences. This section discusses the app verification process in more detail, using the running example of verifying a calculator app.

4.1 Calculator App

The Red Team of the DARPA APAC Program [9] developed several apps, including a calculator that applies the four arithmetic operations to floating-point numbers. Since our JVM bytecode model does not include floating-point numbers yet, we modified the app to operate on integers instead, using Java's normal modular arithmetic. We also slightly simplified the GUI of the app to not use features that are currently not covered by our model. The malware in the app replaces the running result with a random number under certain conditions described later, but we simplified it to return a fixed result of 88888888 instead, because we do not yet model random numbers. These simplifications do not fundamentally change the structure of the app.

4.2 Representation

Our Android model includes a parser, written in ACL2, that turns an app's JVM bytecode class files and XML files into an S-expression-based ACL2 representation usable by our platform model.

A parsed app, with the platform underneath, forms a state machine. The initial state S_0 is defined by our model of app initialization discussed above. Each transition is triggered by a platform-initiated event (e.g., pause app, resume app) or a user-initiated event (e.g., click a button). The deterministic transition function T maps an input event E and a state S to the next state $T(E, S)$; it is lifted to sequences of events by defining $T^*((E_1, \ldots, E_n), S) = T(E_n, \ldots T(E_1, S) \ldots)$, and $T^*(\epsilon, S) = S$, where ϵ is the empty sequence. Our platform model currently supports a single app (state machine) at a time, but can be extended to support multiple apps.

For the calculator app, the state machine has an input event for each calculator button (`0 1 2 3 4 5 6 7 8 9 + - * / = C`) and each app lifecycle event. The state includes a `TextView` GUI object whose content is the string shown on the calculator display. The main correctness theorem for the app says that the contents of the display are always correct, given the sequence of input events supplied to the app so far. We defined an output function O that maps a state S to this display string $O(S)$. Different output functions could be defined for different apps, each extracting from the state the app-specific observables of interest.

4.3 Specification

The execution of the parsed app on the platform model corresponds to a low-level state machine whose states are states of our Android model, as described above, and whose transitions are expressed in terms of the execution of JVM bytecode and API calls. Often a functional specification for an app is naturally expressed as a higher-level state machine, whose states and transitions are defined in user-oriented terms rather than code-oriented terms. The correctness of the code with respect to the specification can then be expressed as a simulation [28] of the high-level machine by the low-level machine.

A state machine specification for the calculator app is sketched in Fig. 1. Each state has a name (in bold, e.g., `value`) and one or more state variables (in italics, e.g., *`val`*); the underlined state variable is the one shown on the calculator display. In each state, *`val`* is the latest result, which is 0 when the calculator starts or when `C` (clear) is entered. In `value-op` and `value-op-value`, *`op`* is the latest operator entered. In `value-op-value`, entering `=` or an operator *`op'`* combines *`val2`* with *`val`* by applying *`op`*, completing the pending operation and replacing the latest result; if *`op'`* was entered, it becomes the latest operator. Figure 1 does not show the expressions assigned to state variables when transitions are taken, e.g., a *`digit`* transition from `value-op-value` to `value-op-value` assigns $10 \times$ *`val2`* $+$ *`digit`* to *`val2`*. Exploiting that 0 is identity for addition,

entering a *digit* in `value` sets *val* to 0, *op* to +, and *val2* to *digit*, as if there were a pending $0 + \ldots$ operation.

We formalized this state machine specification in ACL2. The formalization includes a constant s_0 for the initial state, a deterministic transition function t that maps an input event e and a state s to the next state $t(e, s)$ (and is lifted to t^* over sequences of events, analogous to T^* above), and an output function o that maps a state s to the content of the calculator display $o(s)$.

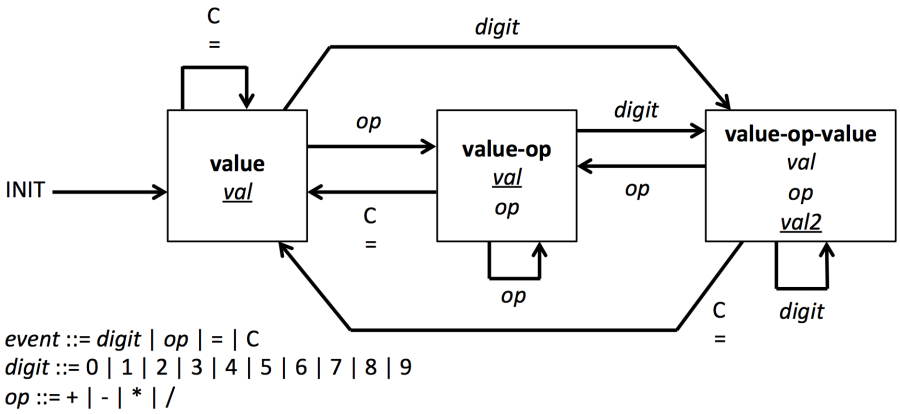

Fig. 1. A state machine specification for the calculator app

4.4 Invariants and Proofs

Often the simulation relation between a low-level and a high-level state machine is defined as an abstraction function [18] from the low-level inputs and states to the high-level inputs and states. For the calculator app, the abstraction function α maps each calculator button press event to the corresponding input in Fig. 1 and each app lifecycle event to no input in Fig. 1; it also maps each app/platform state to a state in Fig. 1.

In our Android platform model, the app/platform state S includes the history of input events. Thus, given an abstraction function to a high-level state machine specification, the correctness of the app with respect to the specification can be expressed as a predicate over the low-level app/platform states. Intuitively, the app's invariant says that the app is in fact in the state that it should be in, given the sequence of inputs seen so far. If $H(S)$ is the history of input events in S, the predicate is $\Omega(S) \equiv [O(S) = o(t^*(\alpha^*(H(S)), s_0))]$, i.e., the observable outputs that result from executing the app's code on the inputs $H(S)$, which take the initial state S_0 to S, are the same that result from running the high-level state machine on the corresponding abstract inputs $\alpha^*(H(S))$, where α^* is the homomorphic lifting of α from events to event sequences. If

Ω includes all the states S reachable from S_0, i.e., if $\Omega(T^*((E_1, \ldots, E_n), S_0))$ holds for every event sequence $E_1, \ldots, E_n$, then the app's code is observationally equivalent to the specification, i.e., it yields the same outputs for the same inputs. For the calculator app, the code is observationally equivalent to Fig. 1. $\Omega(T^*((E_1, \ldots, E_n), S_0))$ is provable by induction if Ω is an invariant, i.e., if Ω holds on the initial state (base case: $\Omega(S_0)$) and is preserved by each transition (induction step: $\Omega(S) \implies \Omega(T(E, S))$). Since Ω alone does not provide a sufficiently strong induction hypothesis, the following invariants are defined, and proved together:

1. A stronger correctness predicate that involves not only outputs (the calculator display) but also states: $\Sigma(S) \equiv [\alpha(S) = t^*(\alpha^*(H(S)), s_0) \wedge O(S) = o(\alpha(S))]$, from which the weaker $\Omega(S)$ is easily proved. While α, t, and S_0 are specific to the app under verification, Σ has the same form for every app whose specification is a state machine with an abstraction function, e.g., the calculator app.
2. Code-level predicates on the app's state, e.g., that a Java `int` field is never negative or is always within a certain range. Formulating these predicates requires an understanding of the app's code, but failed proof attempts in ACL2 often suggest them.
3. Platform-level structural predicates about the Java heap containing the objects that form the app under verification, the Android GUI objects being consistent with the XML files, Java fields having values of the right types, and so on. These constraints are largely boilerplate and we believe that they could be automatically generated at the same time as the app is parsed into its ACL2 representation. For the calculator, we manually defined several predicates of this kind, because their automatic generation is not implemented yet.

Once a sufficiently strong invariant has been defined, proving its establishment in the initial state and preservation by each transition can be carried out by symbolic execution using the ACL2 rewriter. To prove preservation, we start with an arbitrary Android state assumed to satisfy the invariant. We then show that the execution of an arbitrary event results in a state that still satisfies the invariant. The proof naturally splits into cases for each possible allowed event (disallowed events have no effect on the state), and we usually prove each event separately. Some application-specific rewrite rules are often needed (e.g., rules about bit-vector math for the calculator app), and the proofs also use our growing library of rewrite rules about the Android model itself. Otherwise, proofs for simple apps are largely automatic; for the calculator app, the proof corresponding to each button click event is a single line of ACL2 code that invokes our tactic called `def-event-proof`. This tactic unfolds the application of the invariant to the initial state (to expose necessary assumptions for symbolic execution), performs the symbolic execution, often resulting in several cases, and finally, in each case, unfolds the invariant applied to the final state and simplifies the result. In successful proofs, everything simplifies to 'true'.

A key intermediate formula that arises in the proof of the preservation of the invariant is $\alpha(T(E,S)) = t(\alpha(E),\alpha(S))$, i.e., each low-level transition has a corresponding high-level transition—a typical commuting diagram in simulation. If an app's code has no loops (as is the case for the calculator app), ACL2 can automatically prove the invariant's establishment and preservation, provided that an appropriate set of rewrite rules is enabled. The absence of loops is not so uncommon in simple Android apps, where the platform already provides a GUI loop that reads inputs and invokes app code to process them. Verifying apps whose event handlers contain loops is future work and will likely involve formulating and proving appropriate loop invariants; Σ and the other invariants discussed above apply to the platform GUI loop.

We found it convenient to verify the calculator app in two stages. We defined an intermediate state machine whose structure closely resembles the Java code, but without involving any Java or Android concepts. Its states are records whose components correspond to the app's Java fields, and its transitions are defined in terms of record component updates that correspond to the Java code. This intermediate state machine is an abstraction of the code in the ACL2 logic, which in particular does not involve the platform-level structural invariants discussed above. It may be possible to obtain this intermediate machine automatically, using the techniques in [38]. We prove that the app's code simulates the intermediate machine and that the intermediate machine simulates the high-level machine. The two theorems are composed to obtain a proof of correctness of the calculator app with respect to Fig. 1.

4.5 Malware Discovery

The calculator app keeps a count of the operations performed since the last `=` was entered (or since the app started), e.g., after entering `... = 1 + 2 * 3` the count is 2. The malware (in our simplified version of) the app replaces the running result with 88888888 when the count reaches 3. This is functional malware, which does not involve API calls.

We attempted to prove that the calculator app with malware satisfies the specification in Fig. 1. As it should, the verification fails. The output from the failed ACL2 proof exposes the malware: a proof subgoal that cannot proved is that when the operation count is 3, the correct running result is 88888888. In general, failed proof subgoals can expose the conditions that trigger an app's malware and the malicious computations that violate the functional specification.

This is a very simple example of functional malware, which is also fairly easy to detect by the user. However, it is suggestive of more serious, and hard to detect, kinds of functional malware. An example is a military navigation app whose intentional miscalculations send a squad off-course to a dangerous place.

4.6 Functional Bugs

After manually removing the malware from the calculator app, we found two functional bugs in the app that prevented a successful proof. The bugs are also present in the original, unsimplified version.

The operation count is stored in a Java `int`, which wraps around and becomes negative if 2^{31} operations are entered without entering `=`. Since the condition under which the display is updated includes that the count is larger than 1, the display stops updating as the count becomes negative (until it wraps around again to become positive). Since it is impractical to enter 2^{31} operations, this bug has arguably only theoretical significance (some may argue that it is in fact not a bug). Nonetheless, we fixed this bug in the app code.

The other bug may occur in practice: under certain easily achievable conditions, the display is not updated to show the running result. For example, starting the calculator and entering `- 1 2 3 4 5 +` shows 12345 instead of -12345 on the display (the `+` should show the partial result $0 - 12345$, where 0 is the initial display). The details of this bug are unimportant, but are caused by what we regard as an unnecessarily complicated implementation of the calculator: this bug eluded our manual code inspection. While this bug was not malware planted by the Red Team, and is not earth-shattering in its significance, it may be representative of functional malware where a cleverly crafted, non-straightforward implementation may sometimes produce an incorrect result under conditions that cannot be easily detected by manual inspection. After fixing this last bug, we proved the correctness of the app with respect to Fig. 1.

5 Related Work

In [26], JML [20] is used to specify contracts for API and application methods, and the KeY theorem prover [21], which is based on dynamic logic [17], is used to verify that the Java code of those methods satisfies the contracts. Our formal model of the Android API is more comprehensive, e.g., we model callbacks, which are not modeled in [26]. The app specifications in [26] consist of contracts for various app methods, which are implicitly informally "composed" into an overarching correctness argument for the apps. In contrast, our app verification is carried out with respect to an explicit overarching app specification expressed in user-oriented terms (not code-oriented terms like contracts). The translator from Java/JML to KeY in [26] embodies the dynamic logic semantics of Java and JML and is thus a critical component of that approach; in our approach, all the semantics is explicated in ACL2.

In [19], a pencil-and-paper concrete and symbolic operational semantics for Dalvik and for a few Android API methods is defined, and used as the foundation to implement a symbolic executor of Android apps. The symbolic executor is connected to an SMT solver. The tool is shown to infer the conditions under which an example app performs certain privileged actions. Our approach also uses symbolic execution, but our semantics is mechanized inside a theorem prover, and we use ACL2's rewriter for symbolic execution. It is not clear

whether their approach can verify the full functional correctness of apps, due to the use of an SMT solver rather than a more general (but likely less automatic) theorem prover such as ACL2.

In [33], a pencil-and-paper operational semantics for a few Dalvik instructions and a few Android API methods is defined, and a progress property is proved. The paper mentions work in progress on a symbolic executor, but no app verification results are reported. Our Android model is mechanized inside a theorem prover and covers more features of the Android platform.

Other formal models of the Android platform [1,5,13,36] are more abstract than ours, focused on security aspects and properties. These formal models are in a sense complementary to ours: it should be possible to formalize abstraction mappings from our model to those models, ensuring that the security properties of the more abstract models apply to the more concrete model.

Static analysis of app code to help detect malware (e.g., [7,14,16]) is complementary to our approach. It is more automated (e.g., no functional specification is needed) but less precise; it cannot prove deep properties like functional correctness.

In [6], post-conditions of API method calls are calculated from pre-conditions via an algorithm that processes propositional formulas. It may be possible to use our API model and the ACL2 theorem prover for that purpose, which may lead to higher precision in the malware detection tool described in that paper.

Proposals to improve the Android security mechanisms (e.g., [10,30,37]) or to add on-device virtualization (e.g., [22]) require extensions to the platform, which the developers of all the fragmented versions of Android would have to agree on. If implemented, these extensions may prevent certain classes of malware, but not the kind of functional malware that our approach addresses.

Collecting data at run time and analyzing it to detect malware patterns (e.g., [35]) is likely to be more automatic than our approach but may allow malware to execute before it is detected. It also may raise privacy concerns if the analysis is performed off-device.

Dynamic analysis in off-device sandboxes prior to deployment (e.g., [3]) has similar coverage limitations as conventional testing. In addition, some malware may detect when it is being run in an emulator and behave differently than when it is run on a device.

Automatically transforming app code to enforce security policies (e.g., [41]) may affect performance and potentially functionality and may not be agreeable to app developers. This approach may thwart certain classes of malware, but not the kind of functional malware that our approach addresses.

6 Takeaways

App Verification Methodology Many aspects of the app verification work described in Sect. 4 are not specific to the calculator app. We expect that the same proof methodology can apply to a large class of apps:

- Automatically parse the app's code and XML files into a deeply embedded representation inside the theorem prover, obtaining a low-level state machine based on the formal semantics of the JVM and of the Android platform, as in Sect. 4.2.
- Formalize the app's specification as a high-level state machine, expressed in user-oriented terms (not in internal Android-oriented terms), as in Sect. 4.3.
- Define an abstraction function from the low-level state machine to the high-level state machine, as in Sect. 4.4.
- Formulate a sufficiently strong state invariant on the low-level state machine (like Σ in Sect. 4.4) that implies the desired relation between the high-level state machine and the low-level state machine (like Ω in Sect. 4.4). The invariant includes not only simulation conditions, but also code-level invariants and platform-level invariants, as explained in Sect. 4.4.
- Use symbolic execution to prove that the low-level state machine's invariant is established by initialization and preserved by each event.
- If convenient, formalize intermediate state machines (between the low-level one and the high-level one), staging the abstraction functions accordingly. Prove simulations of each machine by the one immediately below it, and finally compose the simulation theorems into one overarching simulation of the high-level state machine by the low-level state machine. As mentioned in Sect. 4.4, for the calculator app we used an intermediate state machine.

State Invariants vs. Trace Invariants By keeping suitable history (e.g., the sequence of events processed so far) in our model of the Android state, we are able to express properties of interest (such as Σ and Ω in Sect. 4.4) as state invariants instead of more complex trace invariants, which involve multiple successive states of execution.

Iterative Invariant Strengthening It may be difficult to formulate a sufficiently strong invariant in one attempt. The first attempt typically results in an invariant that is too weak. However, the failed proof output from ACL2 often readily suggests how to strengthen the invariant. The failed proof output consists of one or more proof subgoals, each consisting of a number of hypotheses and a conclusion. When these hypotheses express some impossible condition (e.g., that an integer variable is outside it possible range of values), the invariant must be strengthened to exclude that impossible condition (e.g., the range of the variable must be part of the invariant). Several iterations may be needed before reaching a sufficiently strong invariant.

Bugs Uncovered by Failed Proof Attempts Bugs in the app (i.e., the fact that the app does not satisfy the specification) are often exposed by failed proof attempts. In some cases, the hypotheses of a failed proof subgoal, when they do not correspond to an impossible situation (i.e., the failed proof is not due to the invariant being too weak), reveal corner cases in which the invariant is broken.

This may indicate either a bug in the app or perhaps a need to reformulate the invariant.

An ACL2 Trick There are cases in which failed ACL2 proof subgoals do not explicitly expose the problem, because the ACL2 rewriter rewrites an untrue conclusion to 'false' and replaces it with the negation of one hypothesis—the untrue conclusion has disappeared from the proof subgoal. This happens, for instance, when attempting to prove that some term x equals a certain constant c, when instead the term equals some other constant c': The goal $x = c$ is rewritten to 'false' and it disappears. To debug this, we can introduce an uninterpreted nullary function f and attempt to prove $x = f()$. The new proof attempt will of course fail, but the rewriter will rewrite x to the correct constant c', displaying the failed proof subgoal $c' = f()$. Then we can revise our original proof attempt to prove $x = c'$ instead.

Android Platform Modeling The Android documentation informally describes the interaction of apps with the Android platform, without explicitly describing most of the internal state of the platform, aside from app lifecycle states and similar aspects. Formalizing the Android platform involves creating an explicit model of the internal Android state. In order to do that, we tried to imagine how the implementation could support the behaviors described in the documentation (e.g., maintain a mapping from resource IDs to references to `View` objects), and defined our state (and transition) model accordingly.

Android API Modeling The large size of the Android API makes its formal modeling challenging. We believe that the best approach to address this challenge is to model the API in a demand-driven fashion, i.e., formalize the API classes and methods as they are needed to verify apps. API methods written entirely in Java need not be explicitly modeled; instead, their code can be symbolically executed along with the app code. However, it may be beneficial to explicitly model API methods that have complex code that may complicate symbolic execution. It should be also noted that, as suggested in [6], typical apps use a relatively small "popular" subset of the Android API: thus, it is not necessary to model most of the Android API in order to verify interesting apps.

7 Conclusion and Future Work

We have described our ongoing work on formally modeling the Android platform and verifying Android apps. Compared to existing research, our Android model has the highest coverage of Android features, and our Android app verification goes deeper to include proofs of full functional correctness. A major motivation for this work is to ensure the absence of functional malware in apps, which other detection approaches to do not address. Our approach can be used to prove deep properties of apps with high assurance.

The proof methodology described in this paper, based on state machines and simulations, can verify a large class of app properties. But the ACL2 logic and our Android model can express other kinds of assertions over the deeply embedded apps. Examples are program-level properties such as the fact that certain API calls are made only under certain conditions and with certain data, which enables much finer distinctions than coarse Android permissions such as `INTERNET`. Other examples are hyperproperties (i.e., predicates over multiple executions) [8], including security policies like non-interference [15], which could express the non-leakage of private user data to network sockets, text messages, and other destinations. To verify these kind of properties, extensions to our proof methodology may be needed, e.g., invariants over multiple states from different execution traces.

We are extending our formal model to cover more Android features and are tackling the verification of larger and more complex apps. We would also like to extend our approach to support reasoning about multiple apps, including their communication via Android's 'intent' mechanism.

Another direction for future research is the modeling and proof of non-functional aspects of apps, e.g., to reason about resource usage or covert channels.

Acknowledgments

This material is based on research sponsored by DARPA under agreement number FA8750-12-X-0110. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

We would also like to thank Garrin Kimmell, James McDonald, and Allen Goldberg for their helpful reviews of this paper.

References

1. Alessandro Armando, Gabriele Costa, and Alessio Merlo. Formal modeling and reasoning about the Android security framework. In *Proc. 7th International Symposium on Trustworthy Global Computing (TGC)*, volume 8191 of *Lecture Notes in Computer Science*, 2013.
2. Alessandro Armando, Alessio Merlo, Mauro Migliardi, and Luca Verderame. Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research*, volume 376 of *IFIP Advances in Information and Communication Technology*. Springer, 2012.
3. Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android application sandbox system for suspicious software detection. In *Proc. 5th International Conference on Malicious and Unwanted Software (Malware)*, 2010.
4. Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

5. Avik Chaudhuri. Language-based security on Android. In *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009.
6. Kevin Zhijie Chen, Noah Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward Wu, Martin Rinard, and Dawn Song. Contextual policy enforcement in Android applications with permission event graphs. In *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.
7. Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.
8. Michael Clarkson and Fred Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
9. DARPA Information Innovation Office. Automated program analysis for cybersecurity (APAC) program. `http://www.darpa.mil/program/automated-program-analysis-for-cybersecurity`.
10. William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2), 2014.
11. William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android security. *IEEE Security & Privacy Magazine*, 7(1), 2009.
12. Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proc. ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.
13. Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing Android's permission system. In *Proc. 17th European Symposium on Research in Computer Security (ESORICS)*, volume 7459, 2012.
14. Adam Fuchs, Avik Chaudhuri, and Jeffrey Foster. SCanDroid: Automated security certication of Android applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, 2009.
15. Joseph Goguen and José Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
16. Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. In *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.
17. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.
18. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
19. Jinseong Jeon, Kristopher Micinski, and Jeffrey Foster. SymDroid: Symbolic execution for Dalvik bytecode. Technical Report CS-TR-5022, University of Maryland, College Park, 2012.
20. The Java Modeling Language (JML). `http://jmlspecs.org`.
21. The KeY project. `http://www.key-project.org`.
22. Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4Android: A generic operating system framework for secure smartphones. In *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.
23. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification – Java SE 8 Edition*. March 2014. `http://docs.oracle.com/javase/specs/jvms/se8/html`.

24. Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you're looking for. DEF CON 18, 2010.
25. Panagiotis Manolios and J Strother Moore. Partial functions in acl2. *Journal of Automated Reasoning*, 31:2003.
26. Masoumeh Al. Haghighi Mobarhan. Formal specification of selected Android core applications and library functions. Master's thesis, Chalmers University of Technology, University of Gothenburg, 2011.
27. John McCarthy. A formal description of a subset of Algol. Technical Report Stanford Artificial Intelligence Project Memo No. 24, Stanford University, 1964.
28. Robin Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Stanford University, 1971.
29. J Moore. Proving Theorems about Java and the JVM with ACL2. `http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-02/index.html`.
30. Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *Proc. 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.
31. Open Handset Alliance. Android Development Resources. `http://developer.android.com`.
32. Open Handset Alliance. Android Open Source Project. `http://source.android.com`.
33. Etienne Payet and Fausto Spoto. An operational semantics for Android activities. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2014.
34. Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A comprehensive security assessment. *IEEE Security & Privacy Magazine*, 8(2), 2010.
35. Ashkan Sharifi Shamili, Christian Bauckhage, and Tansu Alpcan. Malware detection on mobile devices using distributed machine learning. In *Proc. 20th International Conference on Pattern Recognition (ICPR)*, 2011.
36. Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A formal model to analyze the permission authorization and enforcement in the Android framework. In *Proc. IEEE Second International Conference on Social Computing (SOCIALCOM)*, 2010.
37. Stephen Smalley and Robert Craig. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.
38. Eric W. Smith. *Axe: An Automated Formal Equivalence Checking Tool for Programs*. Ph.D. dissertation, Stanford University, 2011.
39. University of Texas at Austin. The ACL2 theorem prover. `http://www.cs.utexas.edu/~moore/acl2`.
40. Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current Android attacks. In *Proc. 5th USENIX Workshop on Offensive Technologies (WOOT)*, 2011.
41. Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *Proc. USENIX Security Symposium*, 2012.