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

Eric Smith and Alessandro Coglio
Kestrel Institute

ACL2 Workshop 2015
(paper presented at VSTTE 2015)

Contributions

- A theorem-proving framework for formal proofs about Android applications.
- Includes an evolving, formal model of (part of) the Android platform.

- Case Study: Verification of a simple calculator app
  - Based on an app produced by a Red Team for DARPA APAC.
  - Proof fails for the malicious / buggy versions.
  - Proof succeeds for correct version.

Motivation

- **Prove functional correctness of Android apps.**

- Also helps detect “functional malware” Ex:
  - give the wrong answer
  - stop working at critical moment
  - lead a platoon off-course

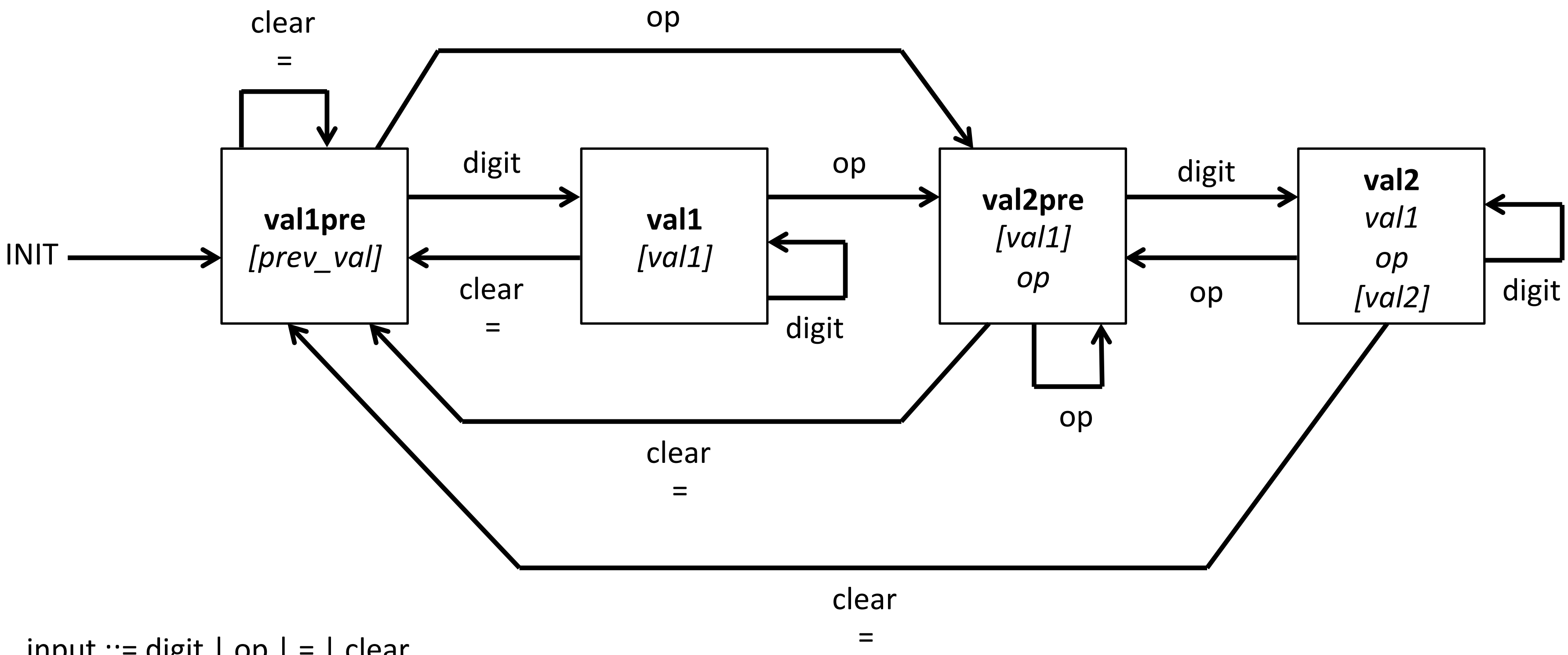
- Malware detection tools are getting good (DARPA APAC)
  - Most data exfiltration can be found
- But no tool available to find functional malware.
  - Not even expressible in most security tools
- And manual inspection can miss subtle behaviors

Outcome

- For incorrect/malicious apps:
  - Proof fails.
  - Bug or malware often indicated by failed proofs.
- For correct/benign apps:
  - Proof gives high assurance proof about app behavior
  - Tells us when we're done: All behaviors rigorously checked

Ex: Correct Behavior of the Calculator App (CalcB)

Formalized as a state machine (`def-state-machine`).



input ::= digit | op | = | clear
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
op ::= + | - | * | /

[…] is the display

Formal Android model

- We developed a formal model of Android
  - Deep embedding of Java Virtual Machine + Android
  - Based on our formal JVM model
  - Models key Android concepts
    - Event-driven
- Model is a formal, executable simulator.
- Reason about the model as it executes the app's bytecode.
  - Proof by symbolic execution (standard technique)
    - Use ACL2 rewriter to repeatedly step and simplify

Formal JVM Model

- Models most Java bytecode instructions (~200)
- JVM state contains: heap, call stack (per thread), static area, loaded classes, monitor table, interned string table, …
- Executable, formal simulator: Shows the effect of each instruction on the JVM state
- Example (IADD instruction):

```
(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)))))))
```

- Many details: exceptions, class initialization, string interning

Formal Android Model 1/2

- Models the state of a single running app (currently)
- Android state contains:
  - JVM state
    - the app’s persistent date (heap and static area)
  - Activity stack
  - Set of currently-allowed events (e.g., button clicks)
  - Manifest (from XML)
  - Layouts (from XML)
  - Current event
  - Various indices
    - View object (e.g., button) -> event listener
    - View name -> resource ID (hex numbers)
    - resource ID -> address of View object
  - API call history (ghost variable)
  - Event history (ghost variable)

Formal Android Model 2/2

- Event-driven:
  - Lifecycle: `(:start), (:resume), (:pause), …`
  - GUI: `(:click “myButton”)`
- Event dispatch:
  - Check if currently allowed (listener registered, no stop before start, etc.)
  - Look up relevant object (e.g., button or activity)
  - Set current event
  - Dispatch to handler : `onClick()`, `onResume()`, …
    - » Execute code
    - » Use models for `super.XXX`() API calls
    - » Code’s effects get recorded in the heap and static area
  - Record API calls made

API Modeling

- Incomplete but growing (driven by the apps we're verifying).

- Sometimes use the code (if available and not too complex):
  - `java.lang.Enum.equals()`
  - `android.app.Activity.setTitle()`
- Sometimes just record and skip
  - `android.telephony.SmsManager.sendTextMessage()`
  - `java.lang.Object.registerNatives()`
- Special handling (fundamental to our model):
  - `setOnClickListener()`
  - `setContentView()`
  - `findViewbyId()`
  - `onStart(), onResume(), …`

Common Proof Methodology

- Formulate Correctness
  - Ex: App matches abstract state machine (state includes history)
  - Ex: Only certain API calls made (don't send text messages)
- Strengthen to an Invariant:
  - Structural invariants: all allowed events, active event listeners, Enum classes, lots of boilerplate (we are automating) …
  - App-specific invariants (e.g., counter never negative)
- Symbolic execution (for each allowed event)
  - start with an *arbitrary* state
  - assume the invariant
  - use symbolic execution (rewriting) to show that running the event handler preserves the invariant
- Top-Level Induction for the Event Loop
  - Since each allowed event preserves the invariant,
  - By induction, conclude that the invariant is preserved for all event sequences.

Automation

- Semi-automatic
- Proof for each calculator button is 1 line. Ex:

```
(def-event-proof (:click "btnPlus") CalcBSimplified6-invariant)
```

- Most work is in formulating the invariant
  - attempt proof and strengthen invariant as needed
- We see lots of things to automate!

Example: Malicious Calculator App

- Malicious Calc:
  - based on an app from a Red Team
  - when number of chained operations is 3, return 88888888
  - this is functional malware
- Attempted proof fails
  - Failed proof shows that the case of interest is when numOps = 3
  - Prover is trying to show that 88888888 is the correct running result
    - Not true and reveals the malware!

Example: Benign Calculator App

Found 2 bugs in “benign” app:

1. Integer overflow in numOps
   - of theoretical interest only
   - after 2^31 chained operations, numOps wraps around and becomes negative
   - display no longer updated until it wraps again

2. Fixed it and tried to prove. But one more issue…
   - Numeric result in display not always updated properly.
   - E.g. starting the calculator (shows “0”) and entering “– 1 2 3 4 +” shows “1234” on the display instead of “-1234”.
   - Corner case eluded informal manual inspection.

Final Proof

- After fixing these two issues, we proved that our calculator app matches the state machine.
- Guarantees that the calculator display always shows the correct numeric result
  - no matter what buttons the user presses
  - no matter what order the buttons are pressed

- We also proved that the calculator only makes allowed API calls (listed in the specification)

Related Work

- To our knowledge, our formal Android model and app proofs are the most detailed to date.
- Lots of related work (see the paper)
- Things that distinguish our approach:
  - Emphasis on Android (not general program verification)
  - Detailed model (not a security/permission abstraction, not a type system)
  - User-level view (vs. checking JML method contracts)
  - Mechanized (not pencil-and-paper)
  - Embedded in a theorem prover (rich logic)
- Most similar:
  - Payet and Spoto: Dalvik model + some APIs, app proofs soon
  - SymDroid (Jeon, Micinski, Foster): symbolic executor + SMT solver

Future Work

- Improve JVM model
  - floating point, Unicode
- Improve Android model
  - more types of events
  - more API calls.
  - track arguments to API calls (URLs visited, phone numbers)
  - Add support for multi-threading, background processes
  - Extend to multi-app system (collusion, etc.)
    - Will need to model Intents
- Handle loops in event handlers
  - lift into logic: turn loops into recursive functions
  - cutpoint proofs of loop invariants

Lessons Learned

- To model Android you have to think like Android
  - Hmmm... To make this work, the platform must keep a map from resource IDs to addresses of View objects. Okay, that has to be part of our state!

- Failed proofs reveals bugs or suggest invariants
  - case that triggers the bug
  - or impossible case (improve invariant)

- Trick: When conclusion rewrites to false, introduce an uninterpreted function
  - Trying to prove X=c1, but X actually equals c2
  - Instead, try to prove X=stub()
  - Prover will fail to prove c2=stub()

- API modeling is hard
  - The Android API is huge!
    - All the APAC teams had this issue
  - Use the code when you can
  - If not (e.g., native methods, fundamental Android methods), write a manual model
  - Do it in a demand-driven fashion

Conclusion

- Formal model of Android (and JVM) in ACL2
- Formal proofs about Android apps
- Using our ACL2 models and proof techniques, we can
  - prove functional correctness of apps
  - find bugs or functional malware

[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.]

Questions?

Extra Slides

Related Work on Android Formal Modeling

- To our knowledge, our formal model of the Android platform is the most detailed to date.
- Other models (e.g. [*]) are more abstract, focused on security aspects.
- It should be possible to formalize abstraction mappings from our model to those models, ensuring that the security properties they prove apply to the detailed model.

[*] Etienne Payet and Fausto Spoto. “An operational semantics for Android activities.” In Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM), 2014.

Related Work on Android App Verification

- To our knowledge, our Android app verification is the most thorough to date.
- Other efforts to mechanically verify functional properties of Android apps at the code [*] level are carried out with respect to code-level specifications for the Java methods that form apps, which are implicitly informally ``composed'' into an overarching correctness argument for the apps.
- Our app verification is carried out with respect to a higher-level specification based directly on user-visible inputs.

References

- Jinseong Jeon, Kristopher Micinski, and Jeffrey Foster.SymDroid: Symbolic execution for Dalvik bytecode. Technical Report CS-TR-5022, University of Maryland, College Park, 2012.
- Etienne Payet and Fausto Spoto. “An operational semantics for Android activities.” In Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM), 2014.
- 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.

Calculator Apps from the Engagements

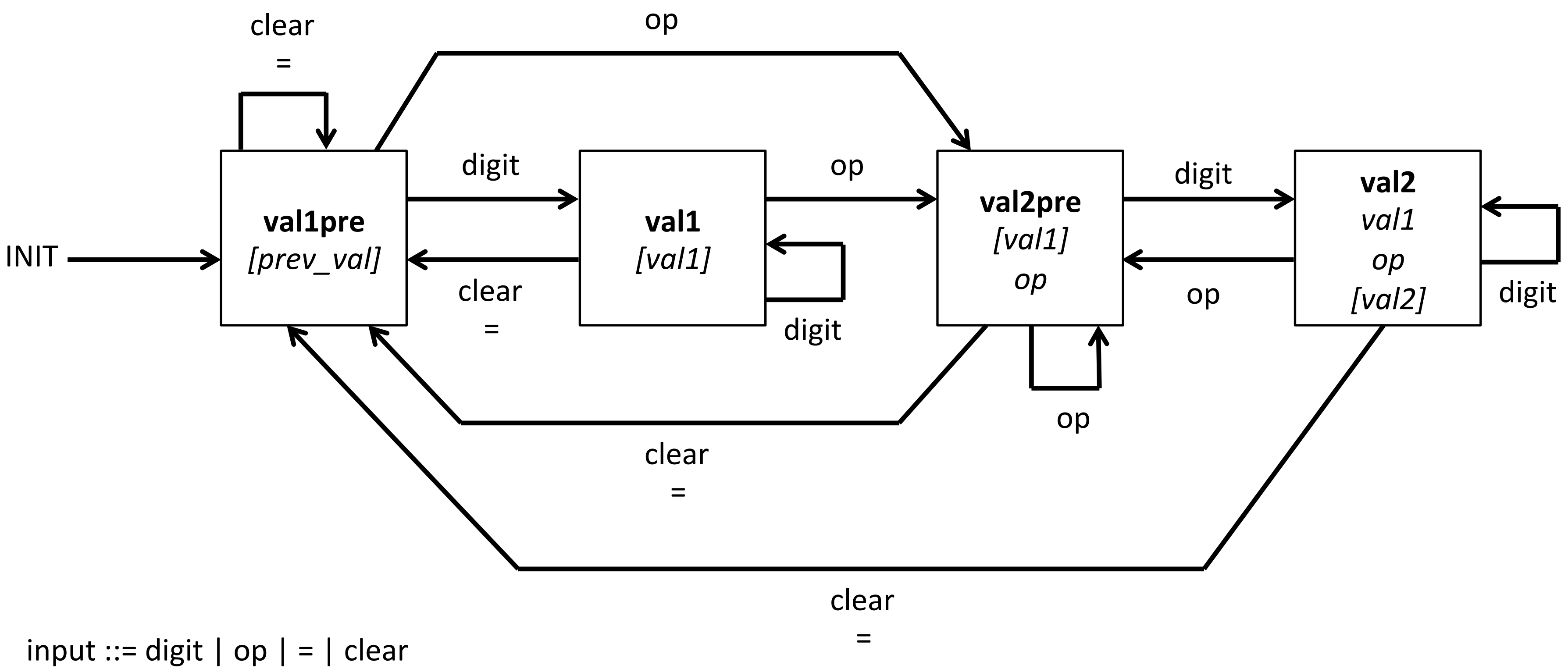
- There are several variants: CalcA, CalcB, …
- They have very similar functionality.
- Their main differences are the presence and nature of malware:
  - Randomly change running result between noon and 1pm.
  - Randomly change running result after 3 consecutive operations (+ - * /) without =.
  - Write to file, then send to a remote server, every operation performed between noon and 1pm.

Our Calculator Apps

- We simplified the engagement apps to work with our current model:
  - We use ints instead of doubles (+ - * / are modular, and / by 0 yields 0), because we do not model doubles yet.
  - A number button modifies the current number directly, instead of appending a char to the display string and then parsing the string into a number, because we do not model the relevant Java API yet.
  - Minor GUI simplifications, e.g. no input from device keyboard (only from buttons) because we do not model the keyboard Android API yet.
  - Malware sets running result to 88888888 after 3 consecutive operations, because we do not model the random-number-generation and time-of-day APIs.
- We made a version of the calculator app without malware, and one with malware.

Formal Functional Specification of the Calculator

We formalized a state machine in ACL2.

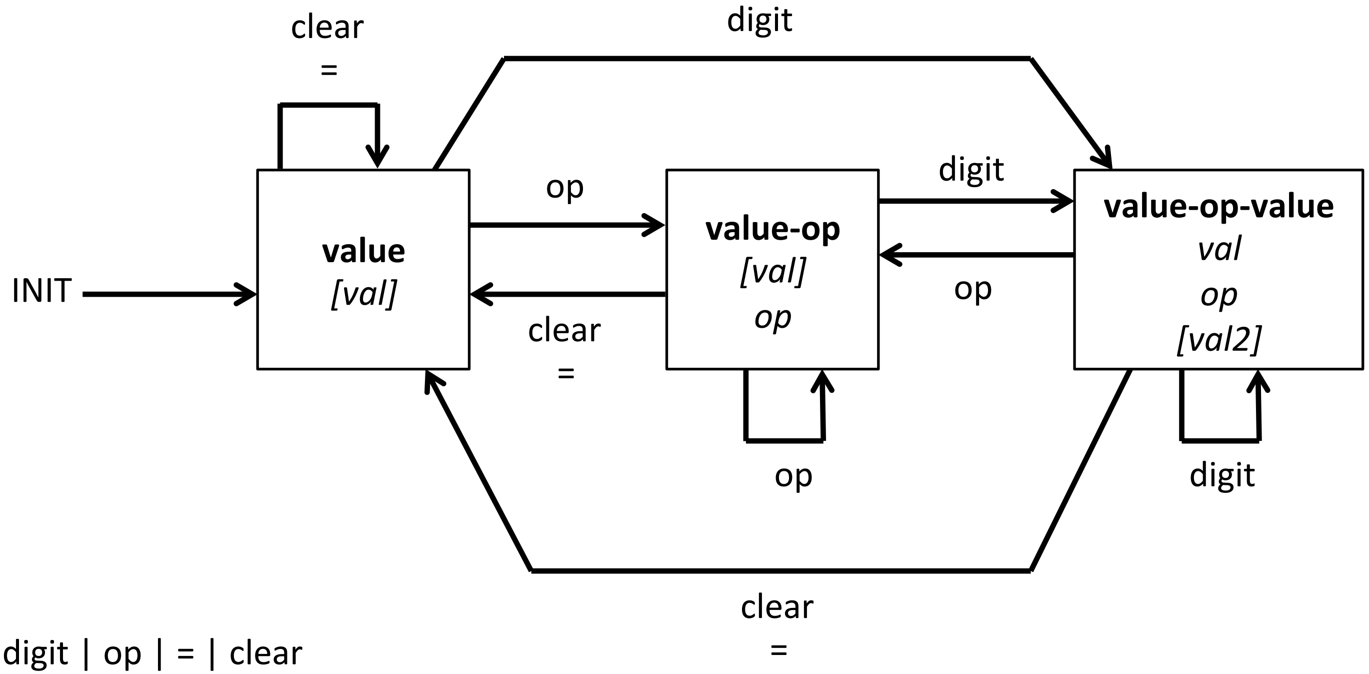


input ::= digit | op | = | clear
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
op ::= + | - | * | /

[…] is the display

Formal Functional Specification of the Calculator (cont'd)

We also formalized a simpler state machine and proved it equivalent to the previous one, in ACL2.



input ::= digit | op | = | clear
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
op ::= + | - | * | /

[…] is the display

Proof Failure Exposes Malware

- We attempted to prove in ACL2 the correctness of the malware calculator app w.r.t. the state machine specification.
- The proof failed, and one of the failed proof subgoals revealed the malware:
  - In the case when the counter of consecutive operations is 3
  - Trying to prove that the running result is 88888888.
  - In general, this kind of failed subgoal shows the conditions on the state variables under which the functional specification is violated.

Proof Process Exposed Functional Bugs in Calculator App without Malware

- We proved in ACL2 that the calculator app without malware satisfies the state machine specification(s).
- But first we had to fix two subtle functional bugs in the (engagement) calculator apps, which we discovered in the course of our proof attempts.

A Minor, “Theoretical” Functional Bug

- After entering 2^{31} operations without =, the display stops updating, until either = is entered or another 2^{31} operations without = are entered.
- This is due to the counter of the number of operations (a Java int) wrapping around.
- Although incurring in this bug is virtually impossible, the app violates the functional specification.
- The specification could be weakened to require the display to be correctly updated only if the number of operations is below a certain value.
- But it is much easier to fix the app to avoid the issue.

A More Severe Functional Bug

- Under certain (easily reachable) conditions, the display is not updated properly.
- E.g. starting the calculator and entering – 8 + shows 8 on the display instead of -8.
- This is due to some corner case in the logic of the app implementation, which looks more complicated than needed (e.g. than a straightforward encoding of the state machine(s)). The corner cases eluded informal manual inspection.

A More Severe Functional Bug (cont'd)

- This functional bug may be representative of a kind of malware triggered by corner cases in the state variables of specially crafted, non-straightforward implementations, that calculate incorrect results under those conditions.
- Static analyzers that abstract away some functionality (e.g. that track information flow) may abstract this kind of malware away.
- Proofs of full functional correctness can uncover this kind of malware.