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

HCSS 2016

Eric Smith and Alessandro Coglio

Kestrel Institute

Kestrel Institute

Contributions

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

- Case Study: Calculator app produced by a Red Team.

Proving Functional Correctness of Android Apps.

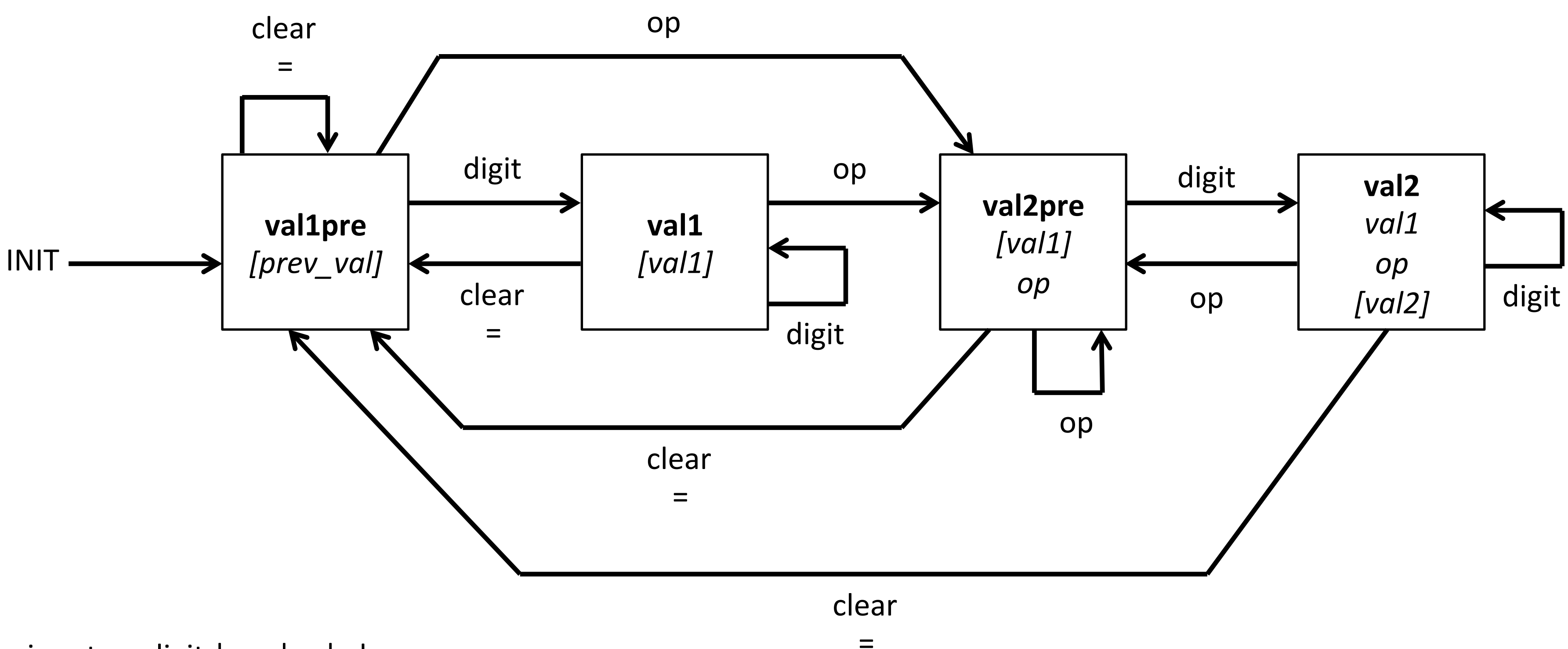
- Not just exfiltration or permission problems.
- Proves correct behavior
- Find bugs
- Finds "functional malware"
  - wrong answer
  - stop working at critical moment
  - lead a platoon off-course
- Few tools can do this
- Better than manual inspection

Benefits

- High assurance app vetting
- For incorrect/malicious apps:
  - Proof fails.
  - Failure often indicates bug / malware
- For correct/benign apps:
  - Proof gives high assurance of correctness
  - Tells us when we're done: All behaviors rigorously checked

Example Spec for Calculator App

Formalized as a state machine.



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

[…] is the display

Our Approach

- Use a theorem prover (ACL2)
- Use a formal model of JVM + Android
  - Deep embedding of Java Virtual Machine
    - Intercept JVM bytecode before translation to Dalvik
  - Model of Android runtime
- Formulate correctness (state machine or predicate)
  - Whatever ACL2 can express
- Prove correctness
  - Common approach:
    - formulate invariant (can refer to history)
    - prove each event preserves invariant

Proofs About Machine Models

- Model is a formal, executable simulator.
- Reason about the model as it executes the code.
  - Proof by symbolic execution.
  - Use ACL2 rewriter to repeatedly step and simplify (standard technique)
  - Conditionals lead to case splits
  - (We have techniques to deal with loops.)

Formal JVM Model

- ~15K lines
- Covers most Java bytecode instructions
- JVM state contains: heap, call stack (per thread), static area, loaded classes, monitor table, interned string table, …
- Models instructions by their effects on the JVM state
- Example (IADD instruction):

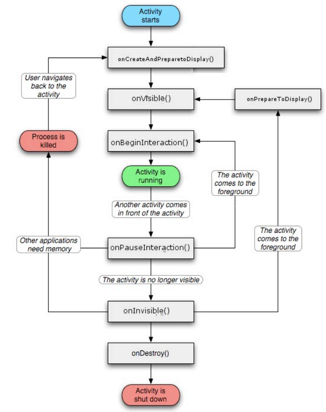
```
(defun execute-IADD (th s)
   (modify th s
     :pc (+ 1 (pc (top-frame th s)))
     :stack (push (bvplus 32 (top (pop (stack (top-frame th s))))
                             (top (stack (top-frame th s))))
                  (pop (pop (stack (top-frame th s)))))))
```

- Many details: exceptions, class initialization, string interning

Formal Android Model

- ~5K lines
- Models the state of a running Android app:
  - JVM state (persistent data in heap and static area)
  - Activity stack
  - Set of currently-allowed events (e.g., button clicks)
  - Manifest (from XML)
  - Layouts (from XML)
  - Event currently being handled
  - Various mappings
    - 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 (cont.)



- 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 the current event
  - Dispatch to handler : `onClick()`, `onResume()`, …
    - Execute handler code
    - Use models for `super.XXX`() API calls
    - Code’s effects get recorded in the heap and static area
  - Record event and API calls made

Android API Model

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

- Try to use the code (if available):
  - `java.lang.Enum.equals()`
  - `android.app.Activity.setTitle()`
- Sometimes handle specially (fundamental to our model):
  - `setOnClickListener()`
  - `setContentView()`
  - `findViewbyId()`
  - `onStart(), onResume(), …`
- Sometimes just record and skip
  - `android.telephony.SmsManager.sendTextMessage()`
  - `java.lang.Object.registerNatives()`

Common Proof Methodology

- Formulate Invariant:
  - Ex: App matches abstract state machine
  - Ex: API calls made so far
  - Structural invariants: active event listeners, Enum classes
  - App-specific invariants (e.g., counter never negative)
- Show it holds initially
- Prove it is preserved (by each allowed event)
  - start with an *arbitrary* state that satisfies the invariant
  - show that running the event handler preserves the invariant
- By induction, show that the invariant is preserved for all event sequences.

Automation

- Not fully automatic …

  … but uses ACL2's highly-automated prover
- Big proofs, lots of cases
- User input for each calculator button is 1 line:

```
(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 boilerplate invariants to automate!

Case Study: Malicious Calculator App

- Based on an app from a Red Team
- When number of chained operations is 3, display 88888888 as the "answer"
- 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
  - Proof failure reveals the malware!

Case Study: Benign Calculator App

Found 2 bugs in “benign” app:

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

2. Missing minus sign in display
   - Ex: Start the calculator (shows “0”) and enter “– 1 2 3 4 +” Display shows “1234” instead of “-1234”.
   - Corner case eluded manual inspection.

Proven Calculator

- Fixed all of these issues
- Proved that our calculator app matches the state machine.
- Guarantees that the calculator always displays the correct numeric result
  - no matter what buttons are pressed

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

Lessons Learned

- To model Android you have to think like Android
  - Hmmm… The platform must map resource IDs to addresses of View objects…
- Failed proofs reveal bugs or suggest invariants
  - Case that triggers the bug
  - Case that should be excluded by the invariant
- Trick: When conclusion rewrites to false, introduce an uninterpreted function
  - Trying to prove X=constant1, but X actually equals constant2
  - Instead, try to prove X=stub()
  - Prover will fail to prove constant2=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

Future Work

- Improve JVM model
  - floating point, Unicode, java.lang.Class
  - run the code for more API methods
- Improve Android model
  - more types of events
  - more API models
  - 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 loops into recursive functions, or
  - use cutpoint proofs for loop invariants

Lots of Related Work
(see the VSTTE15 paper)

- To our knowledge, our formal Android model and app proofs are the most detailed to date.
- 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

Conclusion

- Formal model of Android (and JVM) in ACL2
- Formal proofs about Android apps
- Our approach can
  - prove functional correctness of apps
  - find bugs and functional malware

Paper: Android Platform Modeling and Android App Verification in the ACL2 Theorem Prover. Eric Smith, and Alessandro Coglio. VSTTE 2015 (Springer)

[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?