Flow-Sensitive and yet Sound Static Analysis of Android Applications

Adrien Koutsos, under the supervision of Matteo Maffei

Universität des Saarlandes - CISPA

Goal

Android is today the most popular operating system for mobile phones and tablets, and it boasts the largest application market among all its competitors. The huge number of available applications poses an important security challenge: there are way too many applications to ensure that they go through a timely and thorough security vetting before their publication on the market. Automated analysis tools thus play a critical role in ensuring that security verification are performed on all new applications.

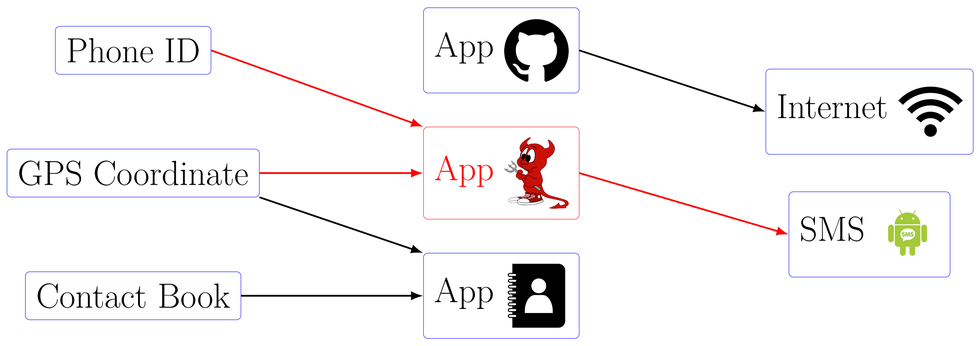

Figure 1: A privacy leak in an Android device

Specificities and Difficulties of Android Analysis

- Application are written in Java, but compiled and shipped in Dalvik, a register based bytecode specific to the Android platform.
- Since Java is an Object Oriented language, static analysis must handle dynamic allocation of objects (which may yield an heap of unbounded size).
- Android applications have a complex life-cycle mechanism and inter-components communications (intents), which is hard to model properly.
- Applications rely heavily on multi-threading.

Contributions

- We extended the semantics of the Dalvik and Android introduced by [2] by adding support for multi-threading (thread spawning and synchronization mechanisms) and exceptions.
- We designed a static analysis for Dalvik bytecode specialized to taint tracking, which handles dynamic memory allocation and thread spawning, and is *flow-sensitive* on the heap while allowing *strong updates*. The analysis is formulated as a set of Horn Clauses soundly over-approximating the semantics of the application to analyse.

Strong Updates vs. Weak Updates

- **Precision:** Strong updates over-write the current abstraction of a variable by a new value, whereas weak-updates add the new value to the set of value over-approximating the variable. Obviously, strong updates allow for a more precise analysis, as it can be seen on the example below.

Code Snippet	Strong Update	Weak Update
`String imei = getId();`	$\widehat{\texttt{imei}} = \{\texttt{id}\}$	$\widehat{\texttt{imei}} = \{\texttt{id}\}$
`imei = obfuscate(imei);`	$\widehat{\texttt{imei}} = \{\texttt{anon}\}$	$\widehat{\texttt{imei}} = \{\texttt{id}\} \cup \{\texttt{anon}\}$
`leak(imei);`	**No Leak**	**Spurious Leak**

Figure 2: Example of an analysis with strong updates, and an analysis with weak updates

- **Soundness:** But strong updates can be unsound in presence of concurrency, because variables may be *shared* between several threads and *interleaved* executions need to be considered. For example, on the code snippet below, a strong update on the abstract memory location over-approximating `imei` will make the analysis miss a leak.

```
String imei = "";
```

Thread 1	Thread 2
`imei = getId();`	`leak(imei);`
`imei = "defaultimei";`	

Method

Our method can be seen as an extension of the *Recency Abstraction* [1] to concurrent settings, and rely on the key observation that a thread can invalidate the approximations computed for another thread only if the two thread share memory. We refine this idea by using two different kind of abstract heap objects:

- *flow-sensitive* abstract objects approximating concrete objects which are guaranteed to be local to a single activity. Moreover, these abstract objects always approximate exactly one concrete object, hence it is sound to perform *strong updates* on them.
- *flow-insensitive* abstract objects used to approximate concrete objects which are either (1) shared between multiple threads, or (2) multiple concrete objects (e.g. produced by a loop). The former is necessary to preserve soundness when sharing occurs, while the later allows to finitely represents heaps of unbounded size. In both case, it is only sound to perform *weak updates*: in case (1), this is a consequence of the interleaved execution of all threads; and in case (2), this follows from the observation that only one of the multiple concrete objects represented by the abstract object is updated at runtime, but the updated abstraction should remain sound for all the concrete objects.

The analysis moves abstract objects from the flow-sensitive abstract heap to its flow-insensitive counterpart whenever one of the two invariants of the flow-sensitive abstract heap may be violated, through a mechanism called *lifting*.

Example

```
public class CoolApp2 extends Activity{
  Contact[] m = new Contact[]();
H(1, {[CoolApp; m ↦ NFS(2)]}) ∧ H(2, [])
  onPause(){
LState3(c ↦ null; 5 ↦ ⊥)
    for(int i=0; i<contacts.length(); i++) {
LState4(c ↦ null; 5 ↦ ⊥)  // i = 0
LState4(c ↦ NFS(5); 5 ↦ ⊥)  // i > 0
      Contact c = contacts.getContact(i);
LState5(c ↦ FS(5); 5 ↦ oc)
      c.phone = anonymise(c.phone);
LState6(c ↦ FS(5); 5 ↦ oc[phone ↦ anon])
      m[i] = c;
LState7(c ↦ NFS(5); 5 ↦ ⊥) ∧ H(5, oc[phone ↦ anon]) ∧ H(2, [NFS(5)])
    }
    send(m, "http://www.cool-apps.com/");
Sink([oc[phone ↦ anon]])
  }
}
```

Figure 3: Analysis of an Application Anonymizing Contact Information

Implementation

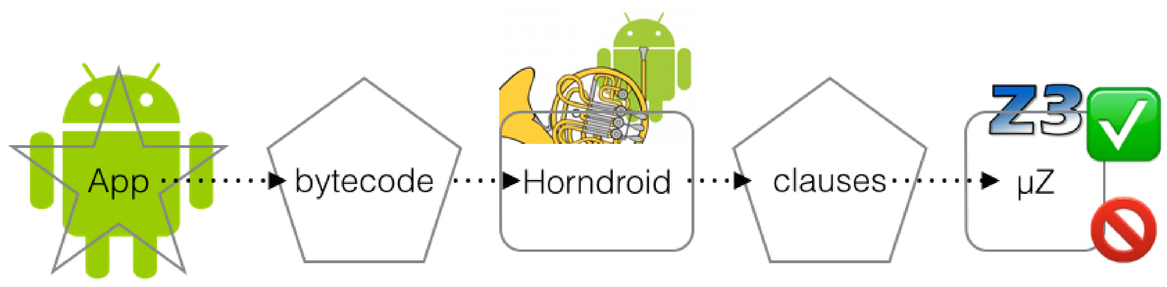

Figure 4: Schema of HornDroid Architecture

We implemented a prototype of our static analysis as an extension of HornDroid [2], a fully automatic static taint tracker for Android applications based on Horn clause resolution. HornDroid encodes the application to analyse as a set of Horn clauses and then uses the SMT solver Z3 to detect flows from private information sources to public sinks. Security verification is performed by the Property-Directed Reachability (PDR) engine μZ implemented in Z3 [3].

	FlowDroid	*AmanDroid*	*DroidSafe*	*HD*	*fsHD*
Precision	0.58	0.74	0.47	0.68	0.79
Average time	22s	11s	2m55s	1s	14s

Figure 5: Comparison with the other state-of-the-art static analysers

References

[1] G. Balakrishnan and T. Reps.
Recency-abstraction for heap-allocated storage.
In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] S. Calzavara, I. Grishchenko, and M. Maffei.
HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving.
In *EuroS&P*. IEEE, 2016.

[3] K. Hoder and N. Bjørner.
Generalized Property Directed Reachability.
In *SAT*, pages 157–171. Springer-Verlag, 2012.

Contact Information

Email address: *adrien.koutsos@ens-cachan.fr*