An Interactive Approach to Mobile App Verification *

Osbert Bastani
Stanford University, USA
obastani@cs.stanford.edu

Saswat Anand
Stanford University, USA
saswat@cs.stanford.edu

Alex Aiken
Stanford University, USA
aiken@cs.stanford.edu

Abstract

Static explicit information flow analysis can help human auditors find malware. We propose a process for eliminating false positive flows due to imprecision in the reachability analysis: the developer provides tests cases, and only tested code is analyzed. Then, the app is instrumented so that executing untested code terminates the app. We use abductive inference to minimize the instrumentation, and interact with the developer to ensure that only unreachable code is instrumented. Our verification process successfully discharges 11 out of the 12 false positives in a corpus of 77 Android apps.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords abductive inference; specifications from tests

1. Introduction

When designing automated code analyses, there is a tradeoff between manual effort discharging false positives and tolerating false negatives. Approaches in practice often rely on precise, dynamic analyses at the expense of possible false negatives, since static analyses produce too many false positives for the user to feasibly examine.

Current static approaches involve the user only at the end (i.e., when displaying results). We believe that a small amount of user input during intermediate steps of the static analysis can significantly reduce the false positive rate. In particular, many false positives are due to imprecision in component analyses: 11 out of 12 false positives in an experiement we performed using static information flow analysis were due to imprecision in the reachability analysis and one was due to imprecision in the alias analysis. Imprecision in these component analyses tend to be more local (compared to information flows) and can be relatively easy for the user to examine. Can we minimally query the user about potential imprecision in component analyses and use the response to the query to discharge false positives?

```
1. void leak(boolean flag, String data) {
2.   if (flag)
3.     sendHTTP(data); }
4. void onCreate() {
5.   leak(false, getLocation()); }
```

Figure 1. An example app $\mathcal{P}_{\text{onCreate}}$.

A static analysis can provide insight to the user about potential imprecision; e.g., insufficient context sensitivity [4], missing models [2, 5], or imprecise reachability information [1]. We use *abductive inference* [3] to formulate a minimal query regarding this potential imprecision (e.g., "is statement s unreachable?"); by minimizing the query, we ensure a reasonable workload for the user. We incorporate the response to the query back into the static analysis and then iteratively make new queries to the user until either we verify the app or no new queries can be formulated.

We implement our approach to discharge false positives due to unreachable code in a static (explicit) information flow analysis; the information flows are used to find possible malware. Our goal is to make queries to a potentially malicious developer [1]. To prevent the developer from falsely claiming "s is unreachable" (so the analysis incorrectly concludes that some information flows are false positives), we enforce such a response by instrumenting the app to terminate if s is reached. Thus, the instrumented app is both consistent with the developer's responses and free of information flows. To prevent the developer from falsely claiming "s is reachable" (to avoid effort), we require that the developer provide a test executing s (proving that s is reachable) with such a response. Using tests has additional benefits: it leverages existing test suites, and tests can be provided to the auditor should the app need to be manually examined.

* This paper is based on [1], and in particular adopts the framework and uses the experimental results of that paper. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

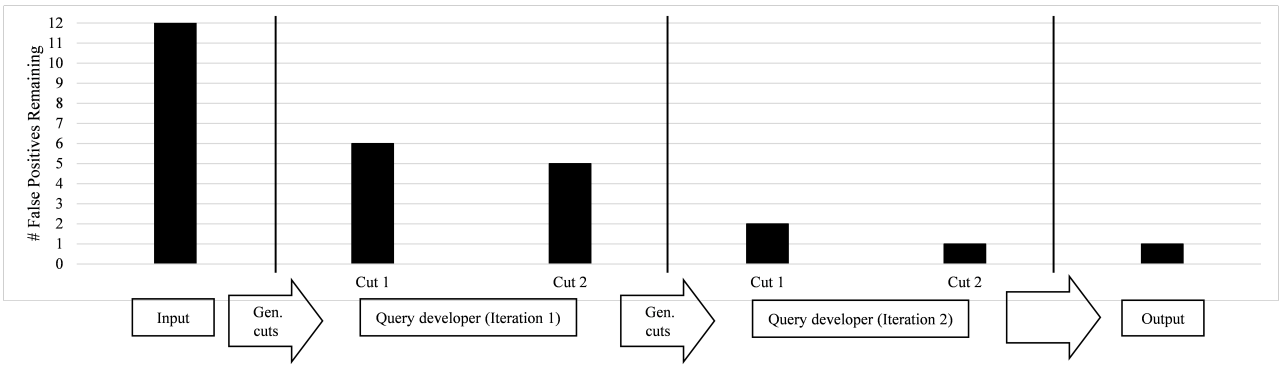

Figure 2. Visualization of the interactive verification process. The steps of the process proceed along the x-axis, and the y-axis describes the number of false positives remaining.

2. Our Approach by Example

Suppose a developer submits the app $\mathcal{P}_{\text{onCreate}}$ shown in Figure 1. First, we run an information flow analysis on $\mathcal{P}_{\text{onCreate}}$; the goal is to verify whether location flows to the Internet. Our analysis removes statements in the program statically shown to be unreachable. Reachability analysis can be very imprecise; for example, unless it is path-sensitive, it would not determine that line 3 is unreachable, in which case we would find that location does flow to the Internet.

Our system searches for a *cut*, which is a subset E of statements that can be removed from $\mathcal{P}_{\text{onCreate}}$ so that the resulting app $\mathcal{P}_{\text{onCreate}} - E$ is free of information flows. The following choices of E are cuts: $E_3 = \{\texttt{3.sendHTTP(data)}\}$ and $E_5 = \{\texttt{5.leak(...)}\}$. However, E_5 is undesirable since line 5 is reachable, so the developer returns a test that executes line 5. Our system executes the test and observes that line 5 is reachable, so it computes a new cut with the constraint that line 5 cannot be cut and returns E_3. The developer accepts this cut, so we instrument $\mathcal{P}_{\text{onCreate}}$ to terminate if execution reaches line 3. Because E_3 is a cut, the instrumented app cannot leak location data, and because E_3 is unreachable, $\mathcal{P}_{\text{onCreate}} - E$ is semantically equivalent to $\mathcal{P}_{\text{onCreate}}$ and the instrumentation incurs no runtime overhead.

We compute cuts by reducing the problem to an integer linear program (where the objective is to minimize the size of the cut); see [1] for details.

3. Experimental Results

We demonstrate the effectiveness of our approach by verifying a corpus of 77 Android apps to be free of malicious information flows; our focus is on eliminating the false positives found by our static information flow analysis. We play the role of the developer (determining reachable statements by reading the bytecode). To reduce the number of iterations, we query the developer on two cuts for each iteration instead of one; only one of the two cuts must be valid (i.e., only contain unreachable statements).

In Figure 2, we show two iterations of the interactive process. The black bars show the cumulative number of false positives remaining to be discharged at each point in the process. We start with 12 false positives. In the first iteration, for 7 out of the 12 false positives, at least one of the two cuts was valid, so we accept the cut (allowing the static analysis to discharge the false positive). We discharge 4 additional false positives in the second iteration. The one remaining false positive is due to imprecise aliasing rather than unreachable code, so no valid cut exists; the auditor must manually examine this false positive.

4. Conclusion

We have described an approach for interacting with an auditor (or developer) to eliminate false positives in static analysis. The static analysis knows the (local) sources of imprecision that may cause false positive (global) information flows. By making a small, carefully selected query to the auditor regarding these local sources of imprecision, we can discharge a significant number of false positive information flows (92% in our experiments). We believe that our approach to handling false positives can significantly improve the usability of static analysis.

References

[1] O. Bastani, S. Anand, A. Aiken. Interactively verifying absence of explicit information flows in Android apps. In *OOPSLA*, 2015.

[2] O. Bastani, S. Anand, A. Aiken. Specification inference using context-free language reachability. In *POPL*, 2015.

[3] I. Dillig, T. Dillig, A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, 2012.

[4] X. Zhang, R. Mangal, R. Grigore, M. Naik, H. Yang. On abstraction refinement for program analyses in Datalog. In *PLDI*, 2014.

[5] H. Zhu, T. Dillig, I. Dillig. Automated inference of library specifications for source-sink property verification. In *APLAS*, 2013.