Interactively Verifying Absence of Explicit Information Flows in Android Apps

Osbert Bastani, Saswat Anand, and Alex Aiken

Stanford University

OOPSLA 2015

Problem

- Google Play Store
  - > 1 million apps on the store

- Lots of malware submitted
  - Information leaks
  - SMS Fraud
  - Ransomware

Information Flow Analysis

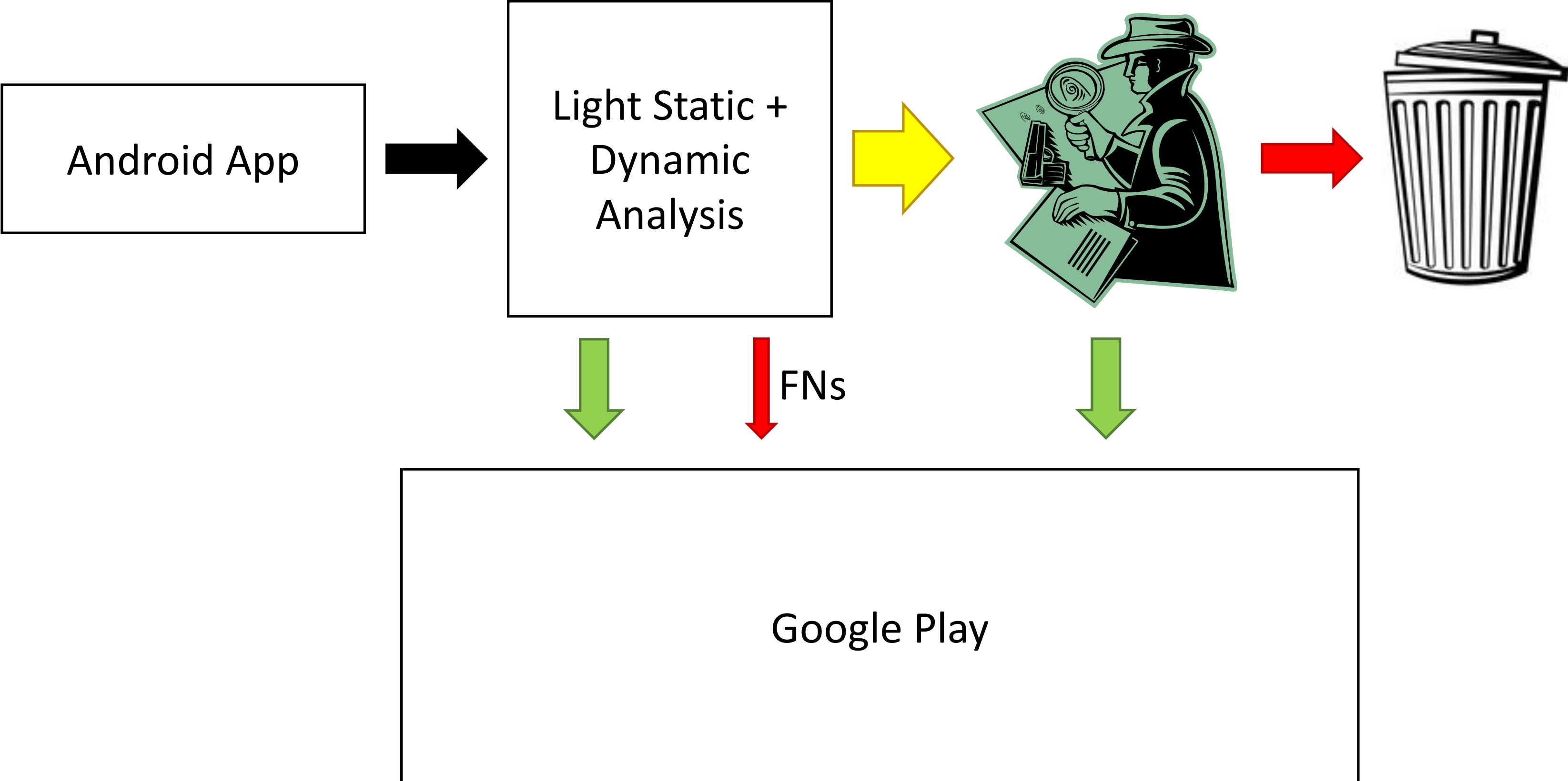
- Finding Android malware using source to sink flows

Information leak:	location	flows to	Internet
SMS Fraud:	phone #	used in	SMS send
Ransomware:	network packets	encrypt	files

Standard Audits

- No static information flow
  - Too many false positives
- Light-weight static analysis
  - Dynamic code loading
  - Calls to undocumented APIs
- Dynamic analysis
  - Information flows

Standard Audits



Dead Code

- Dead code can cause false positive information flows
  - Global property (e.g., method with no caller)

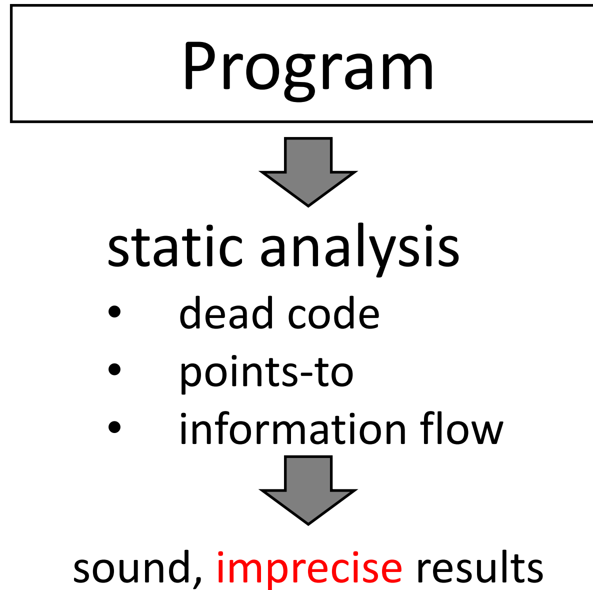
- Examples
  - Leaks in 3rd party libraries
  - Conservative assumptions about potential callbacks

Key Issue

- Hard to understand **someone else's** code
  - No source code!
  - Obfuscation

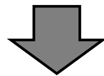
- Can we shift work to **developer**?

Developer Queries



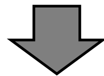
Developer Queries

Program



static analysis

- dead code
- points-to
- information flow

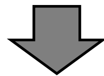


sound, imprecise results

potential dead code

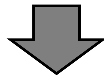
Developer Queries

Program



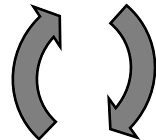
static analysis

- dead code
- points-to
- information flow



sound, imprecise results

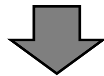
yes/no



potential dead code

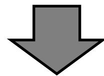
Developer Queries

Program



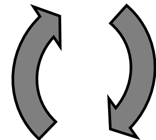
static analysis

- dead code
- points-to
- information flow



sound, precise results

yes/no

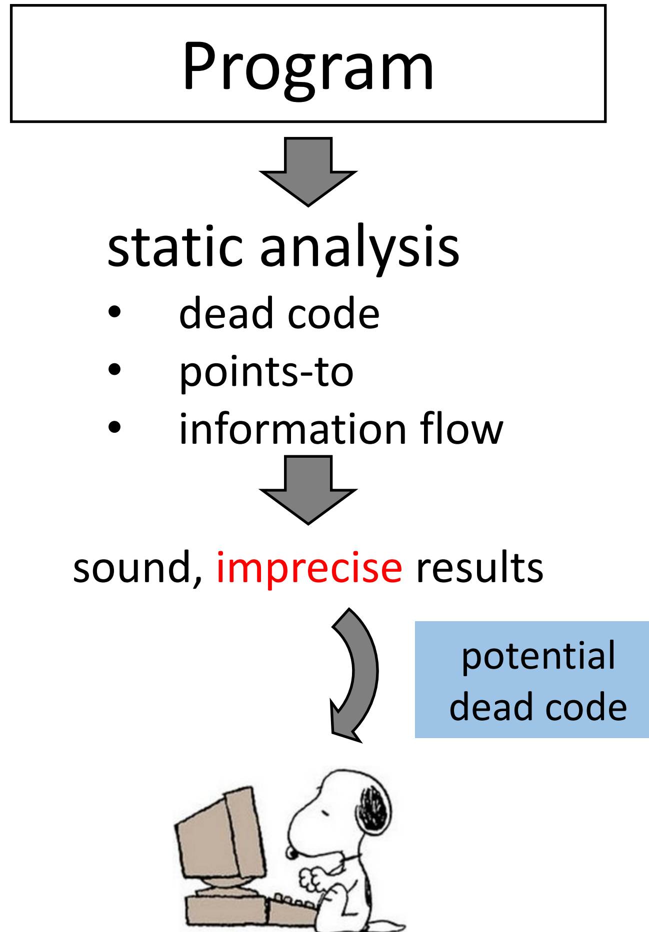


potential dead code

Problem

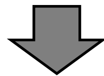
- Developer may be **adversarial**
- Solution: **Enforce** response

Developer Queries

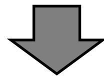


Developer Queries

Program

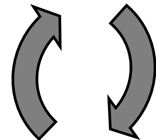


static analysis
- dead code
- points-to
- information flow



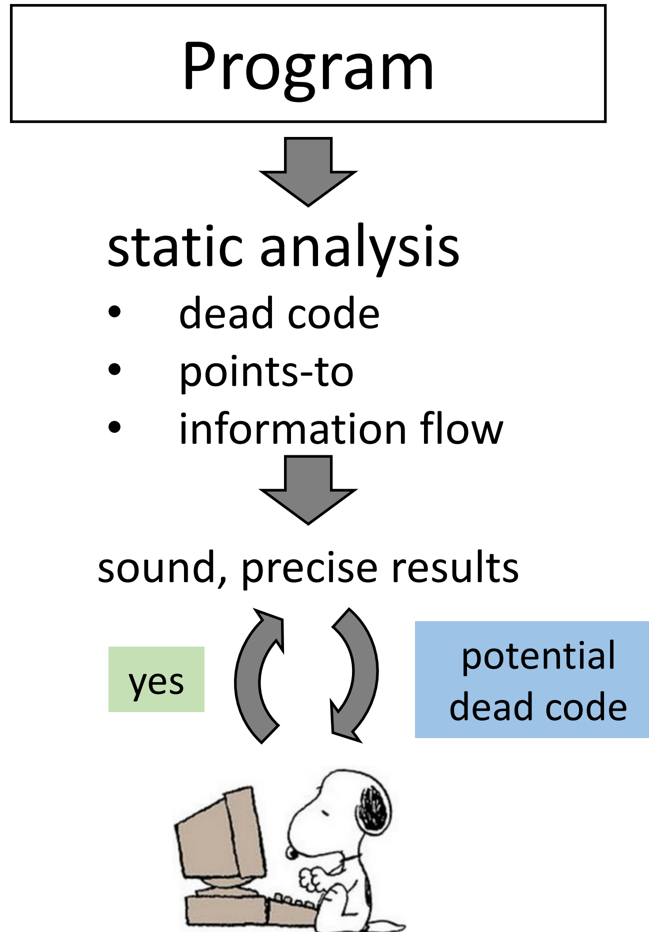
sound, imprecise results

yes

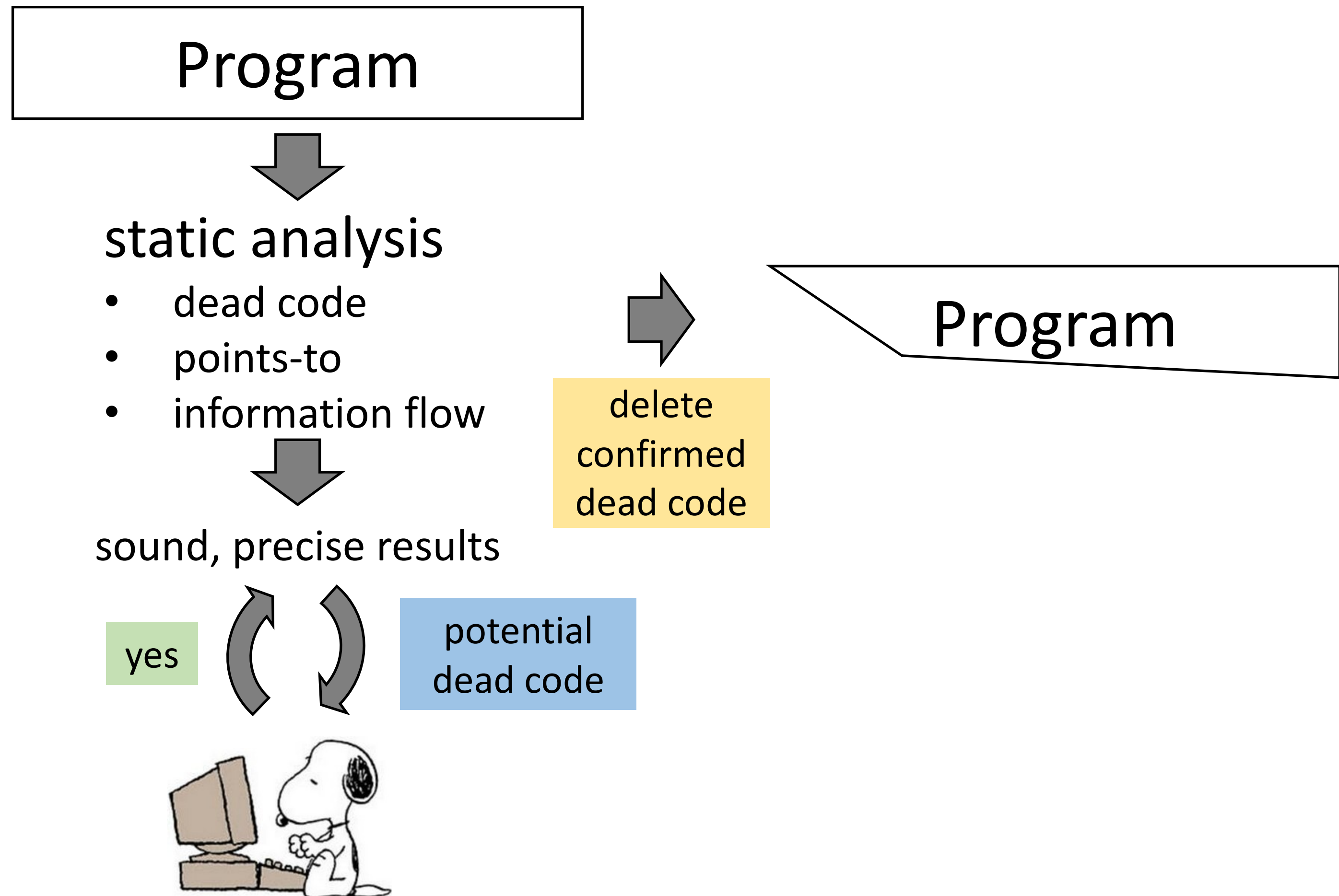


potential dead code

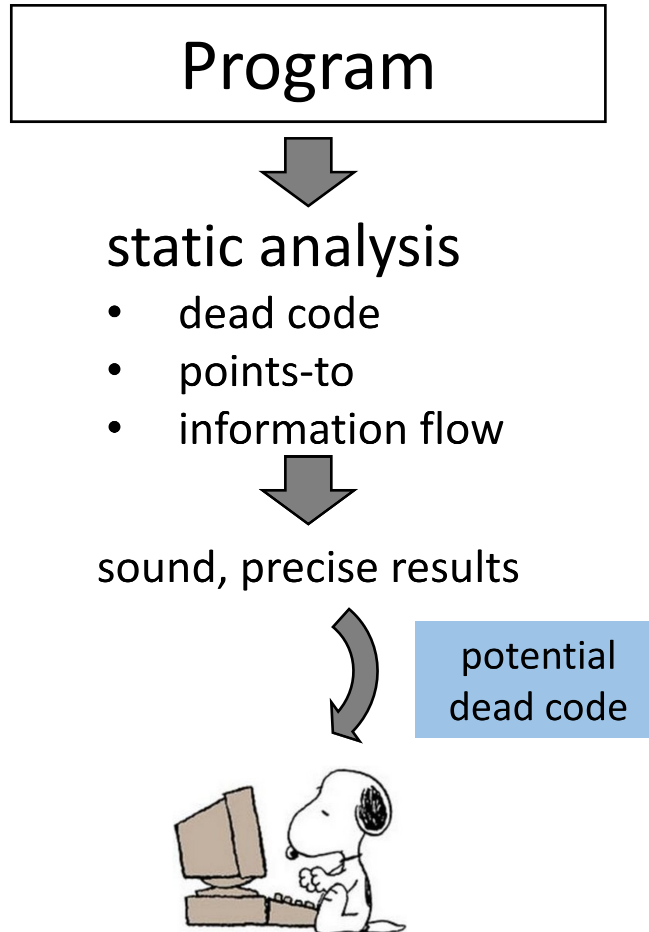
Developer Queries



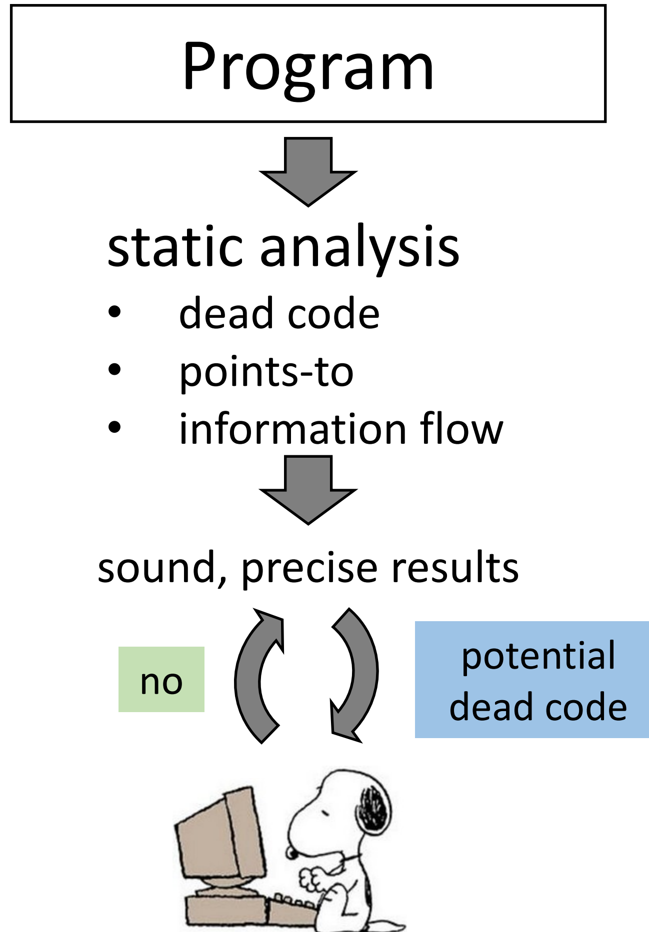
Developer Queries



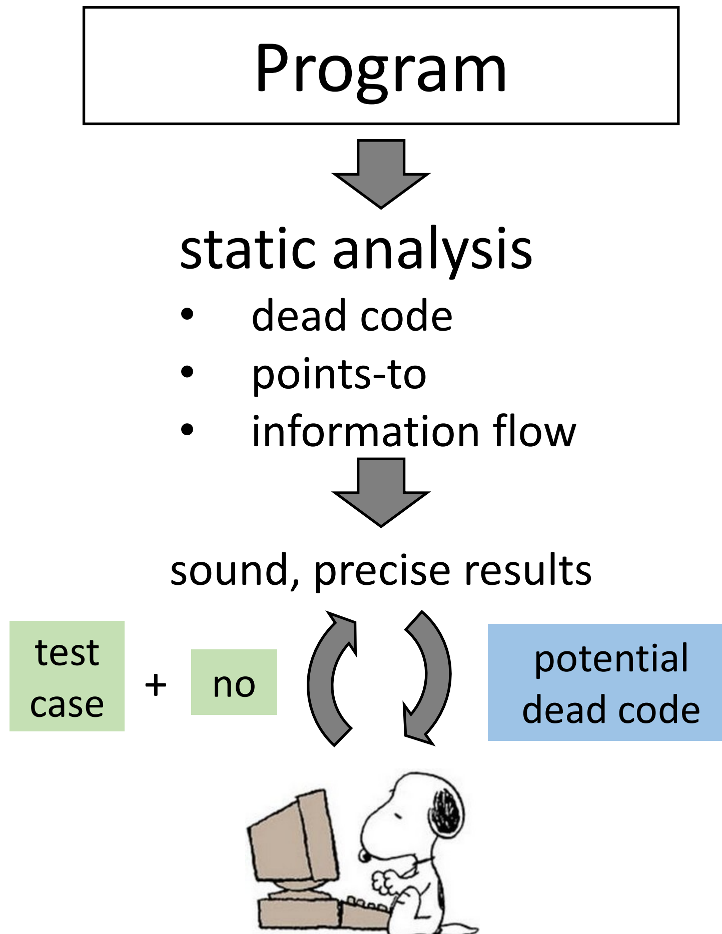
Developer Queries



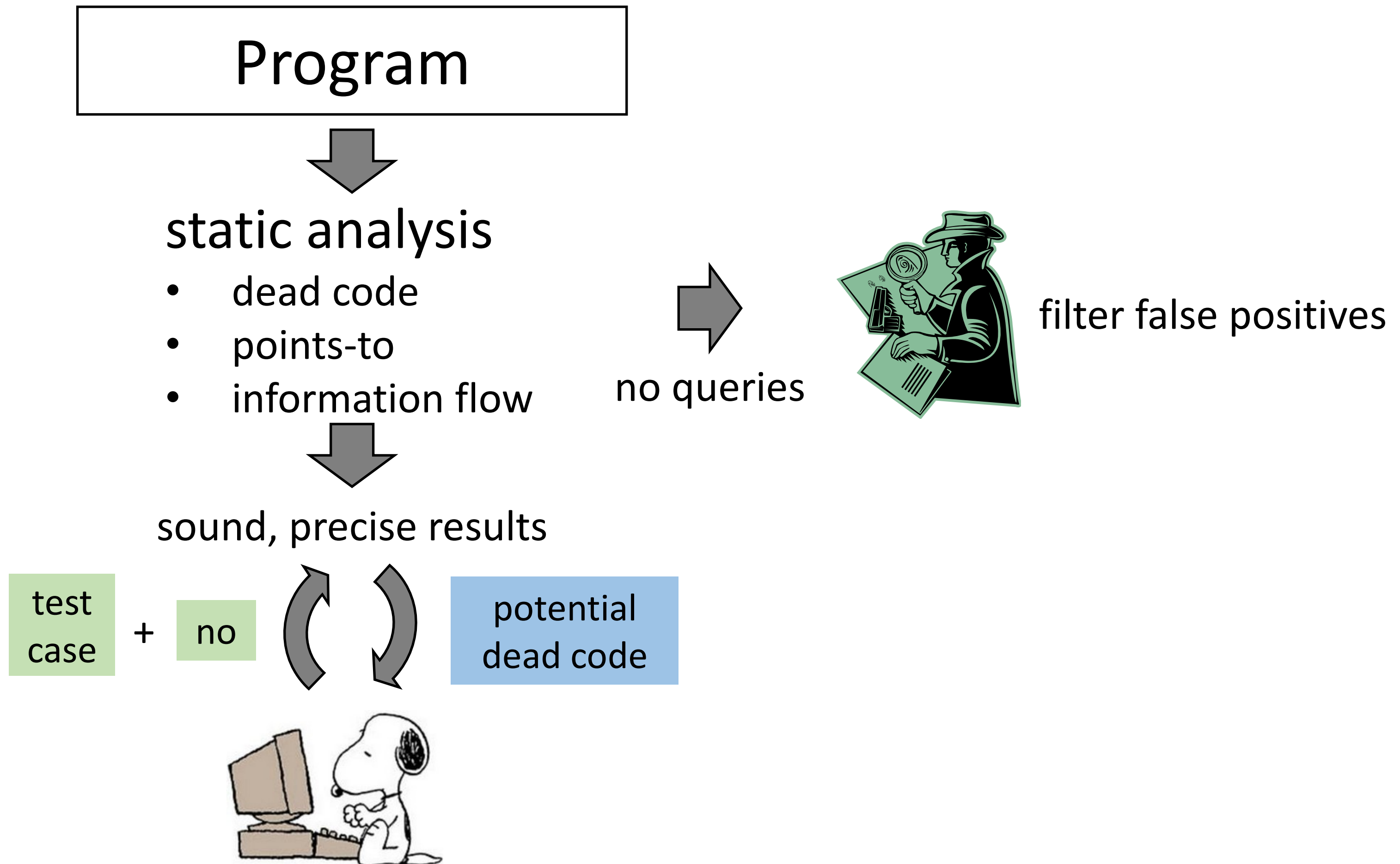
Developer Queries



Developer Queries



Developer Queries

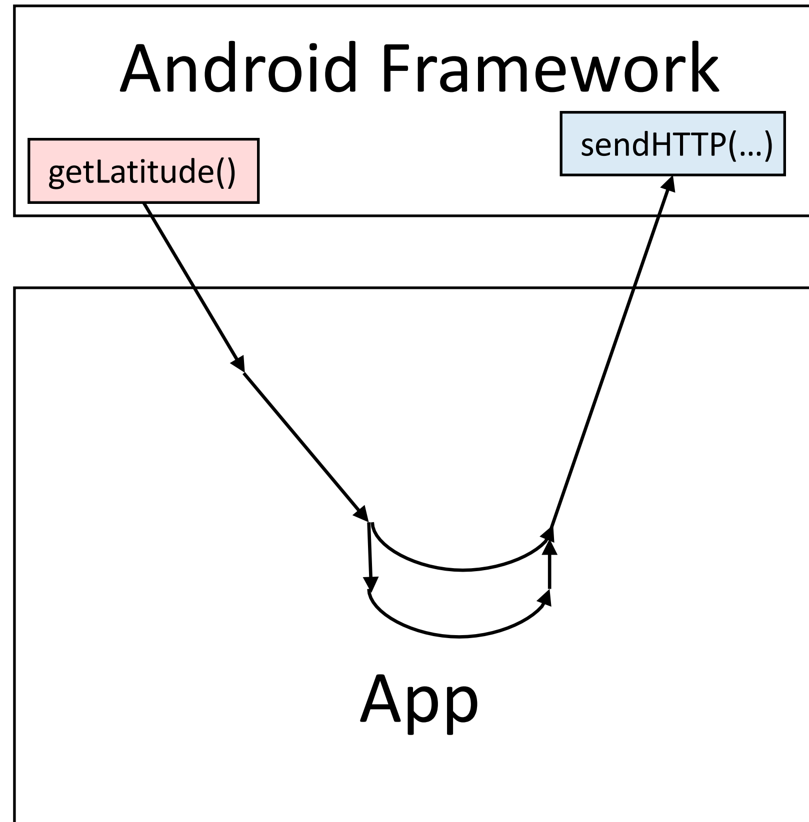


Developer Queries: Cuts

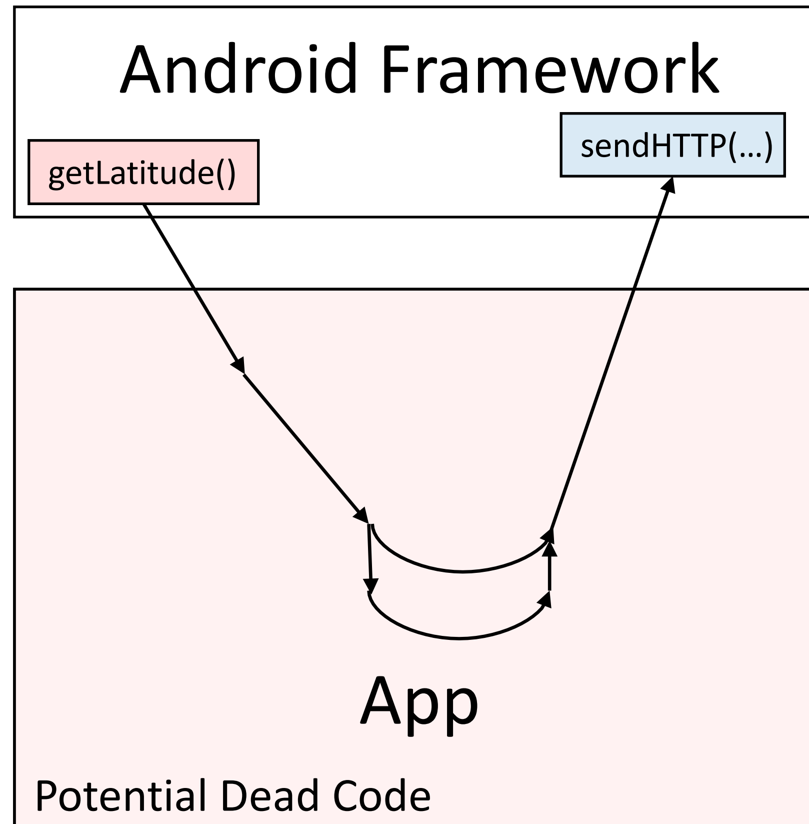
- *Cut*
  - Code that is (potentially) dead
  - Removing the code breaks (potential) information flows

- *Valid cut*: Developer confirms that the cut is dead

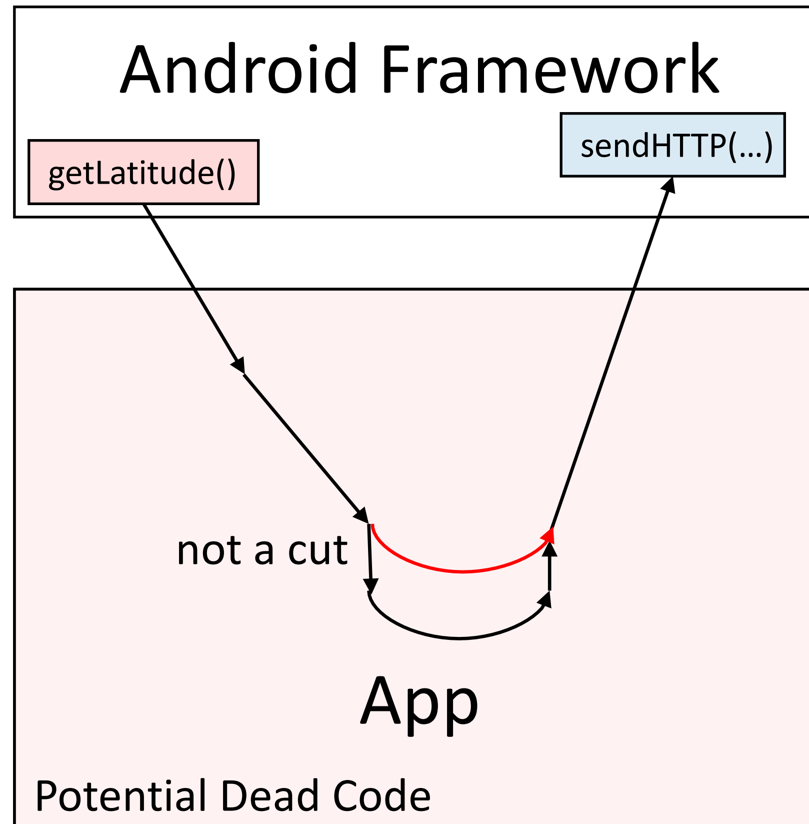
Developer Queries: Cuts



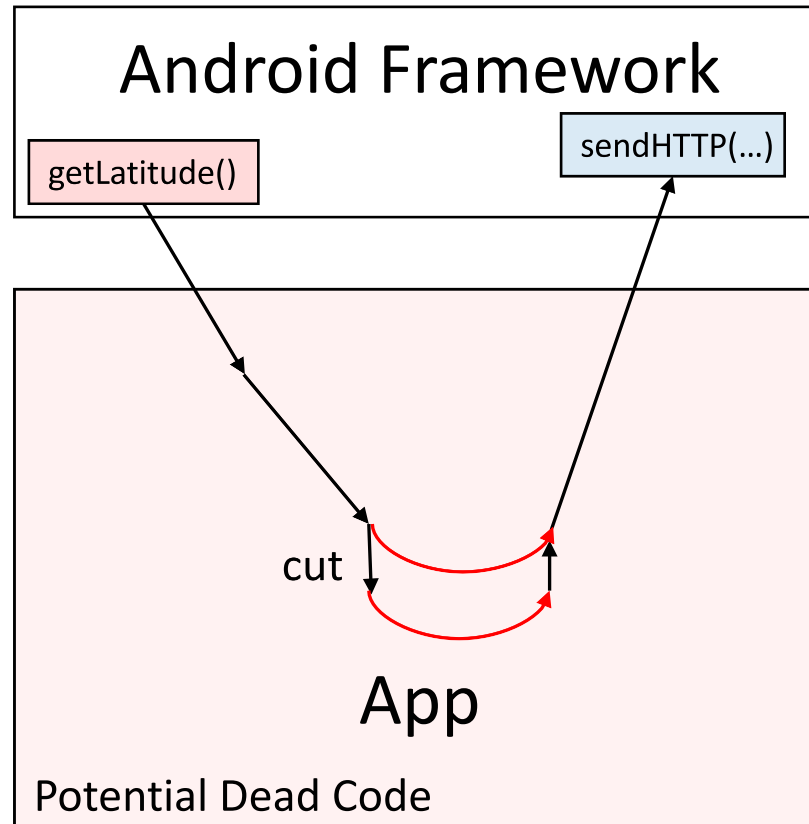
Developer Queries: Cuts



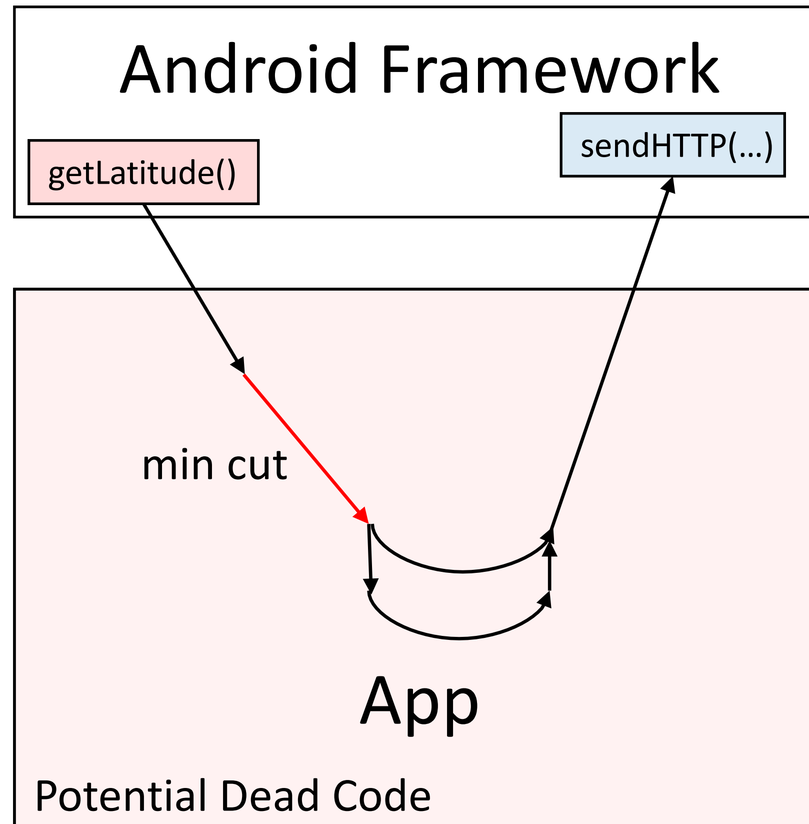
Developer Queries: Cuts



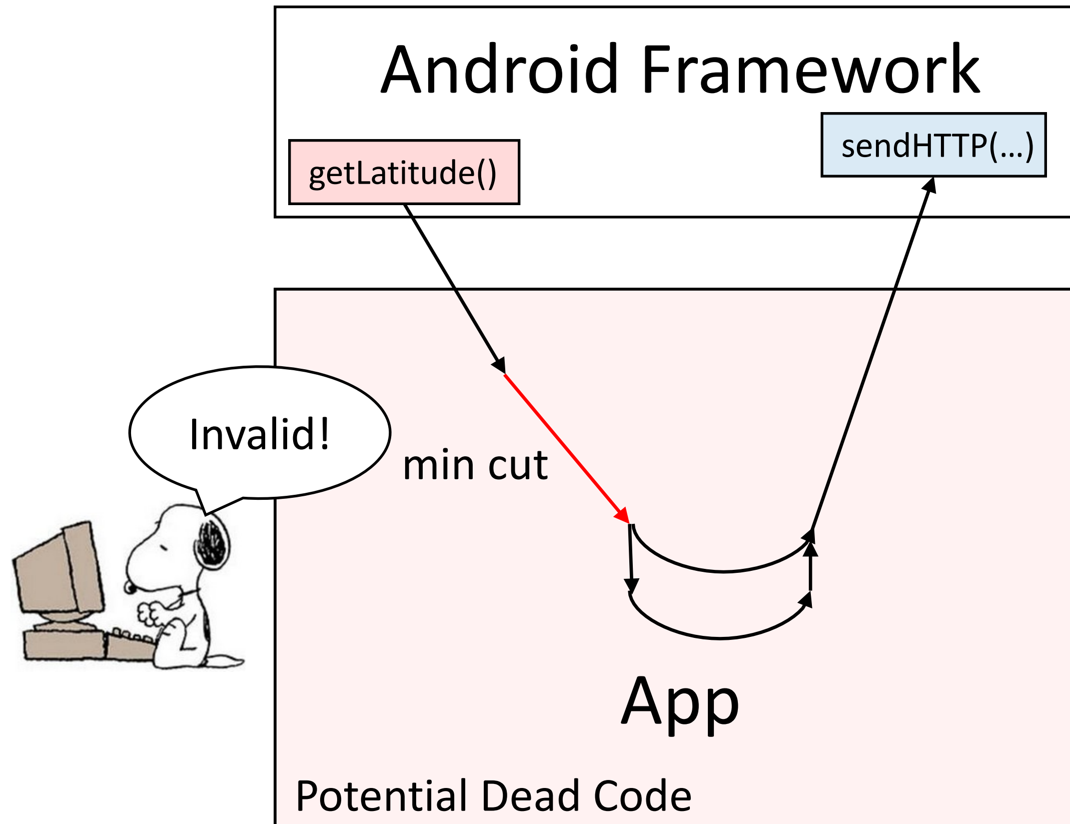
Developer Queries: Cuts



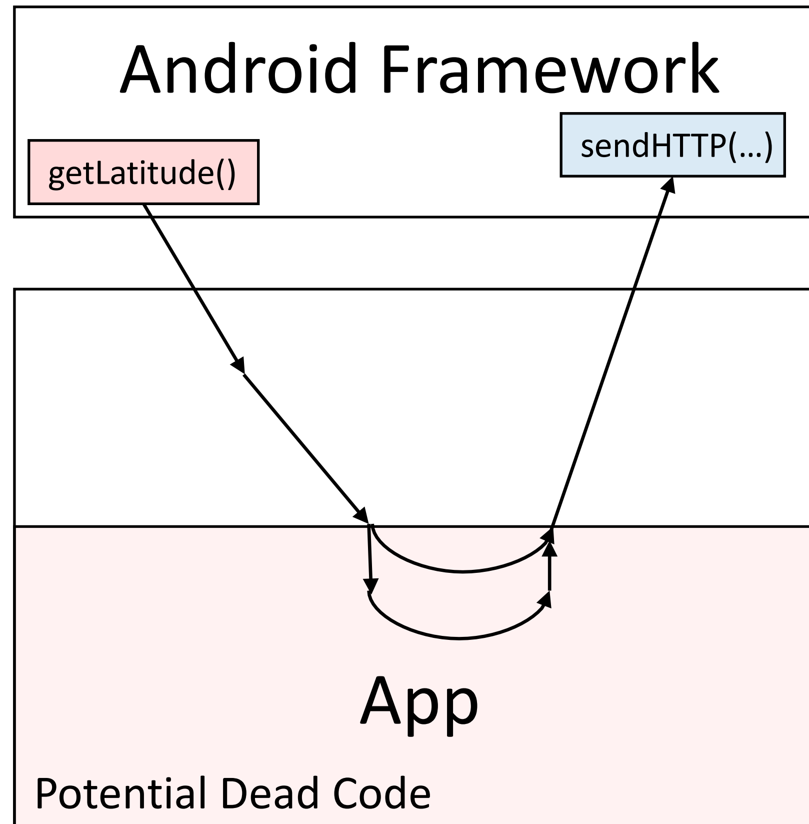
Developer Queries: Cuts



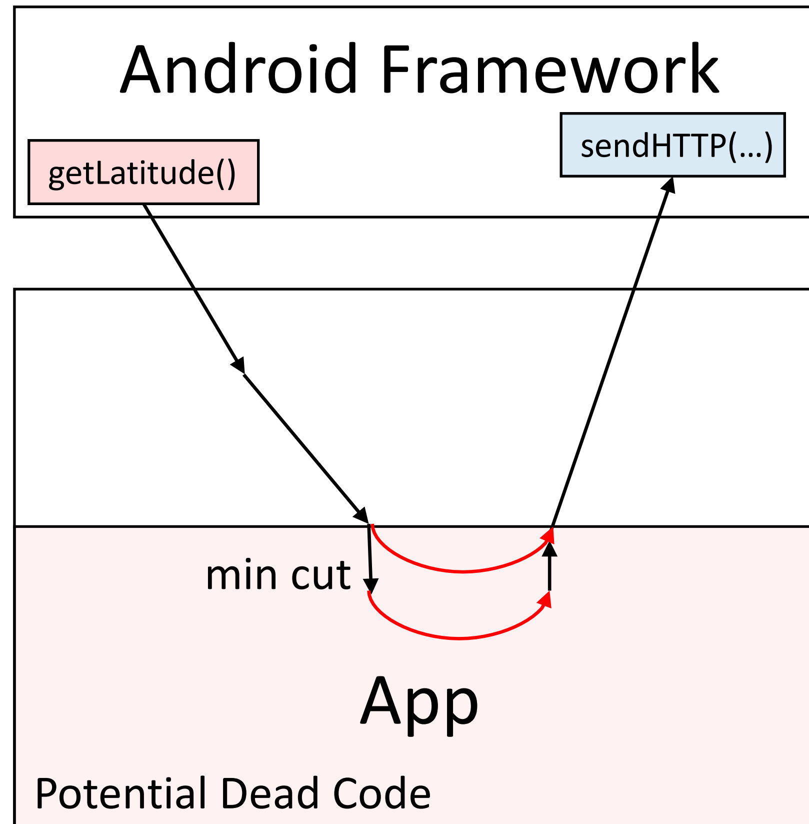
Developer Queries: Cuts



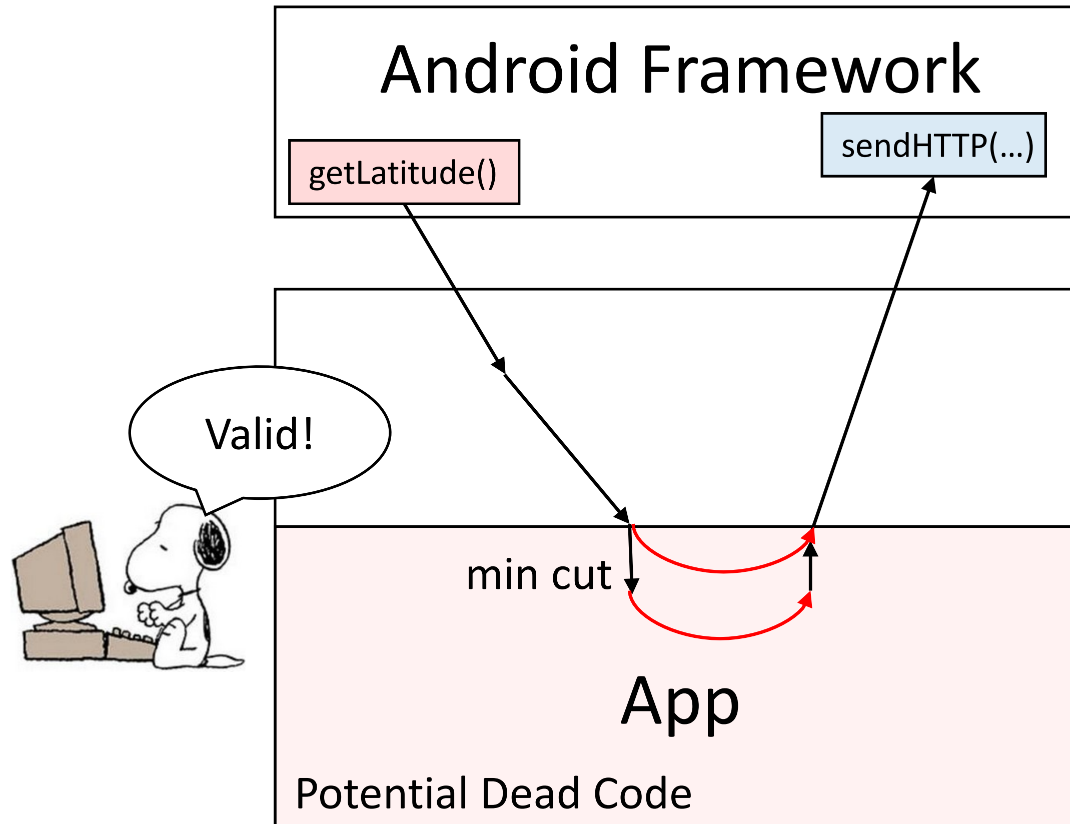
Developer Queries: Cuts



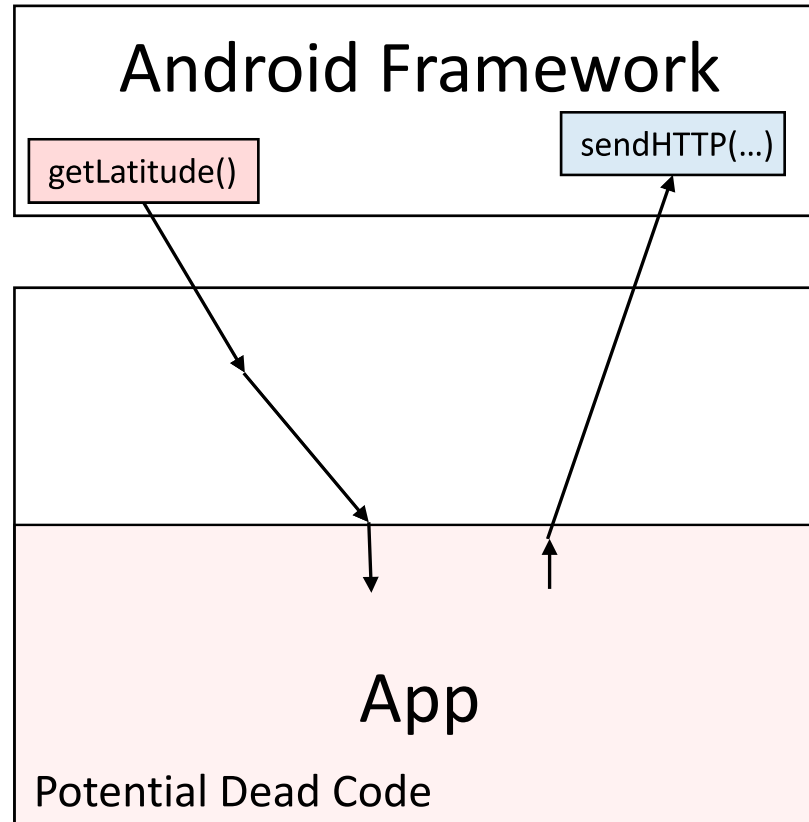
Developer Queries: Cuts



Developer Queries: Cuts



Developer Queries: Cuts



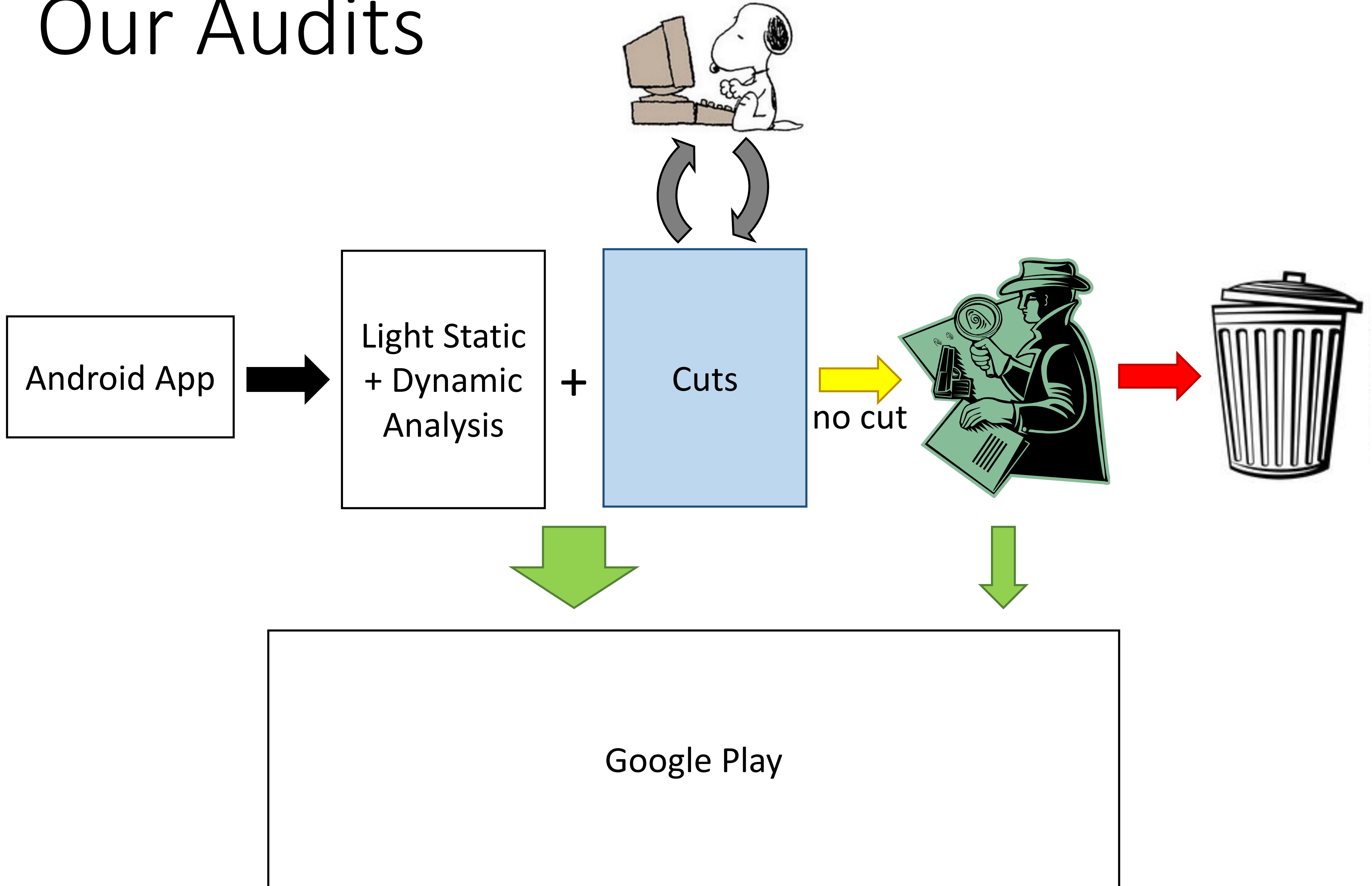
Developer Queries

- Developer query
  - List of statements in cut
- Developer response
  - Test case that executes reachable statements in the cut
- Why tests?
  - Verifiable
  - Developers routinely write tests
  - Can seed with dynamic analysis
  - Aid auditor

Interactive Verification

- **Step 1:** Sound static analysis
- **Step 2:** Find cut and query developer
- **Step 3:** Update potential dead code and repeat
- **Step 4:** *Delete* valid cut

Our Audits



Finding Valid Cuts

Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```

Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```

LOCATION

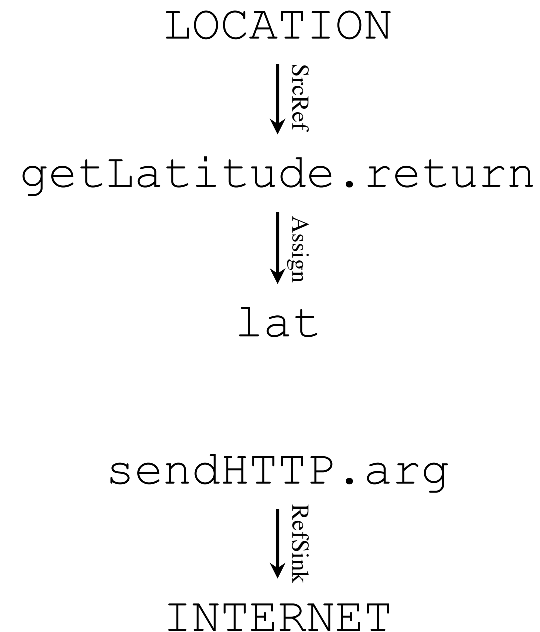
getLatitude.return

sendHTTP.arg

INTERNET

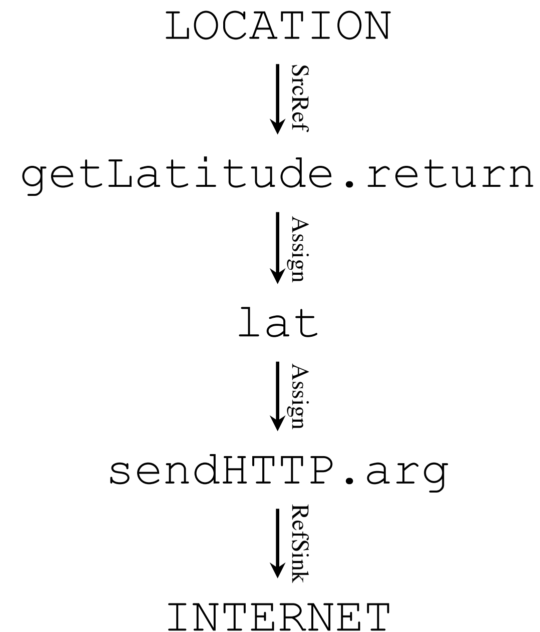
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



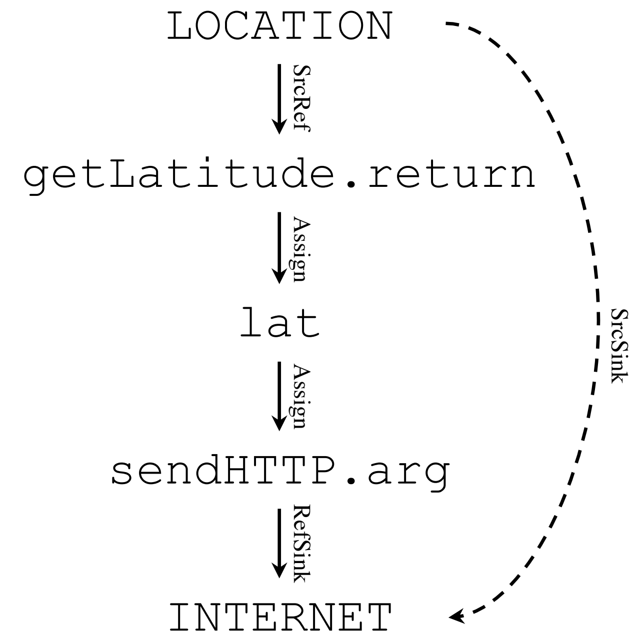
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



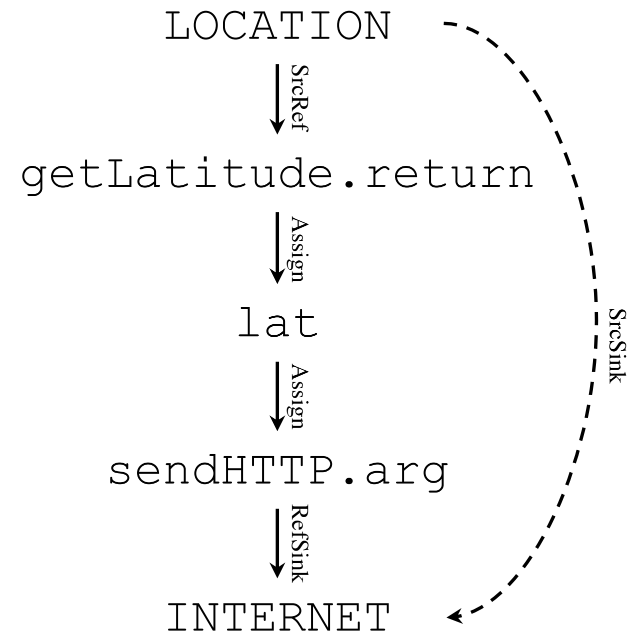
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



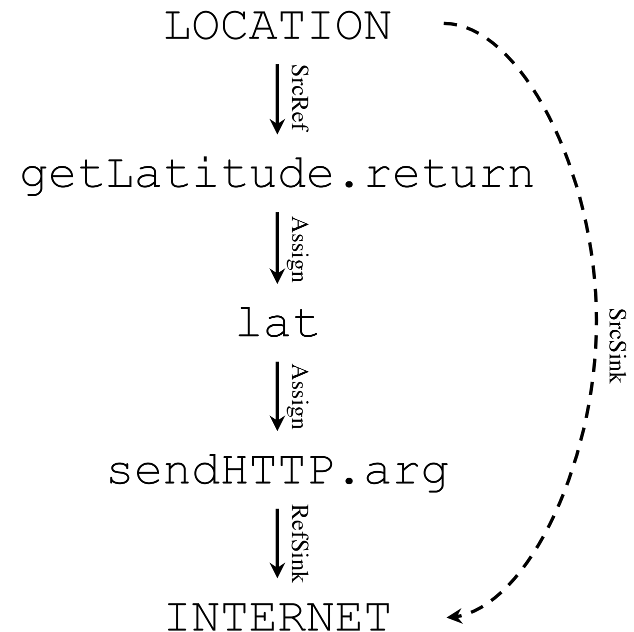
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



Finding Valid Cuts

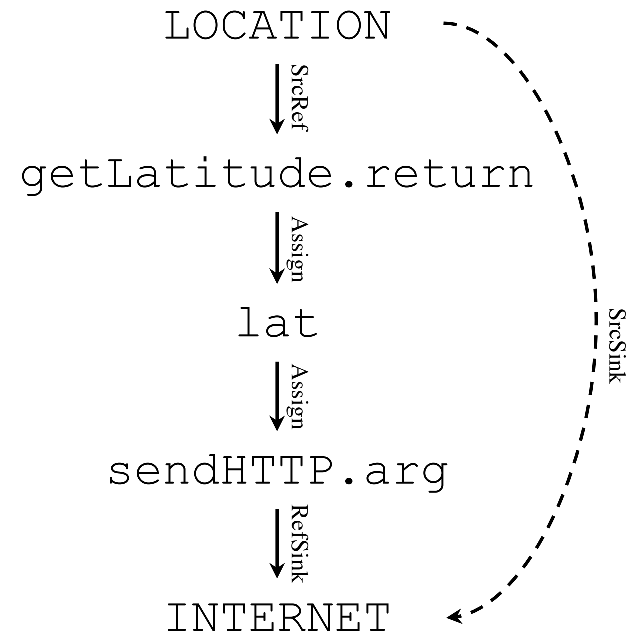
```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```

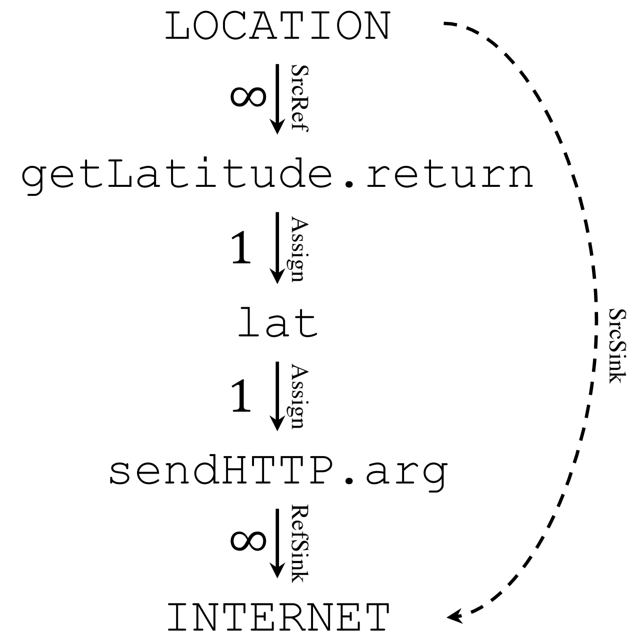
potentially dead



Finding Valid Cuts

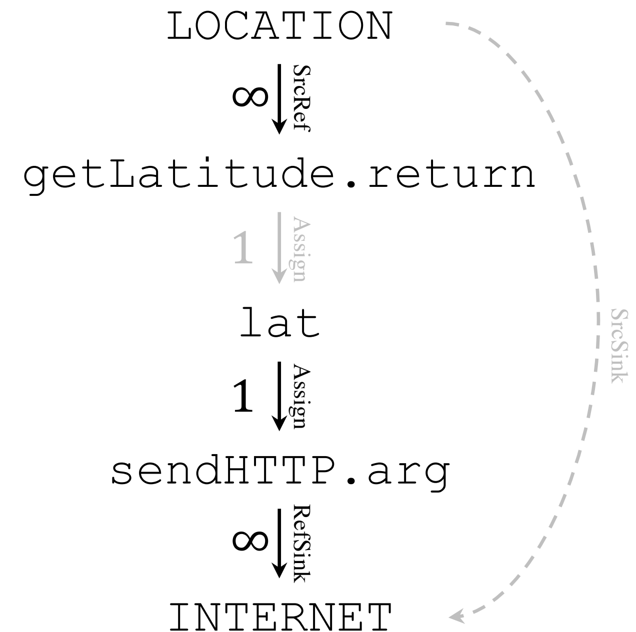
```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```

potentially dead



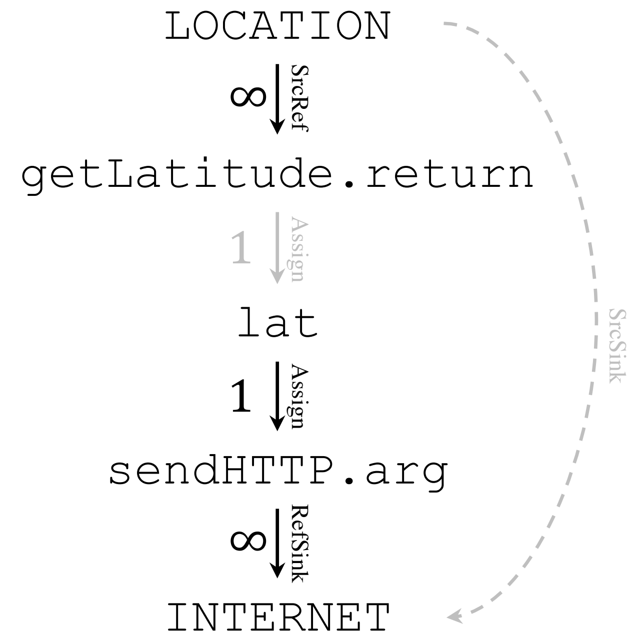
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



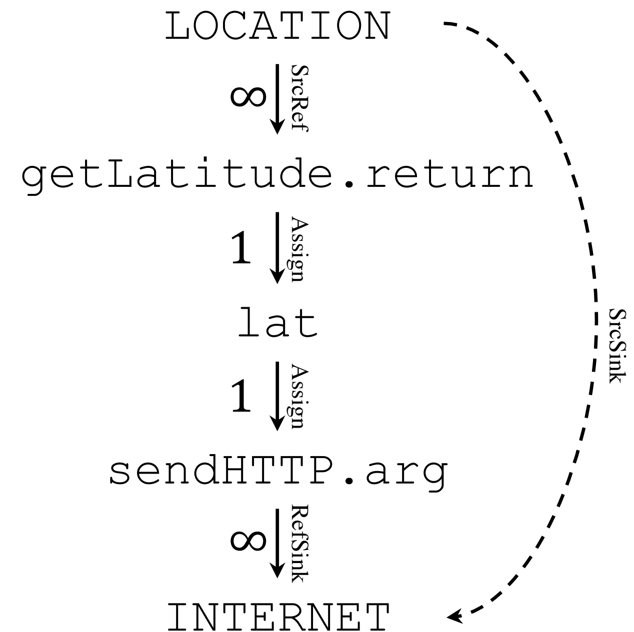
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



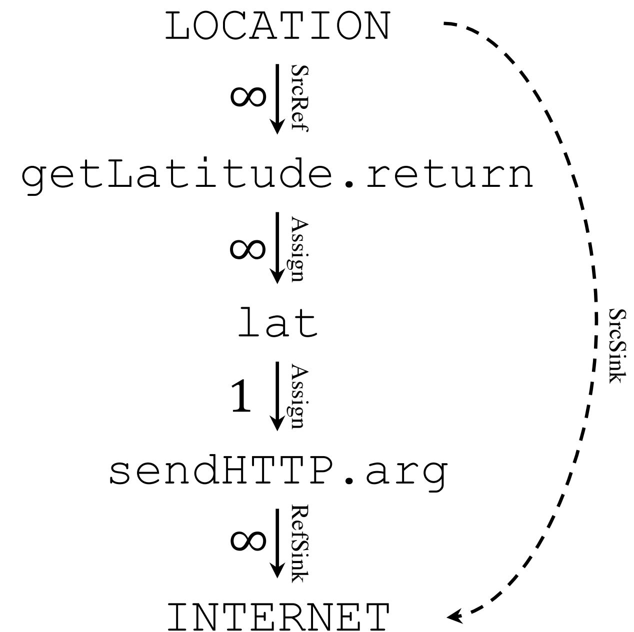
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



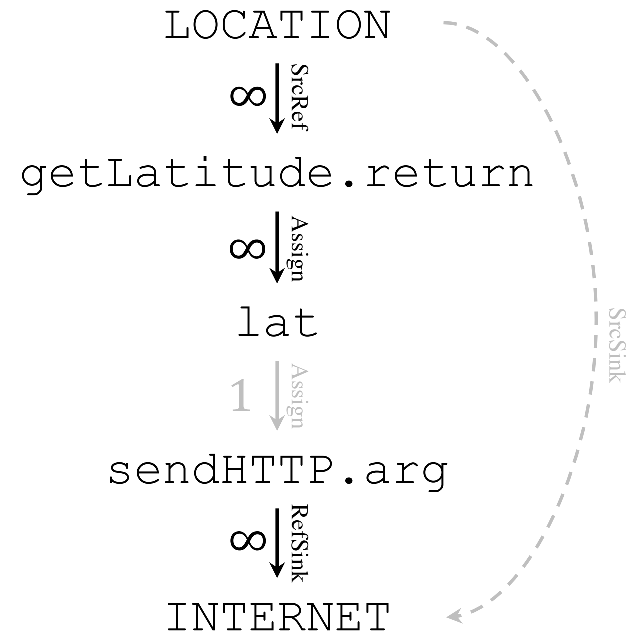
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



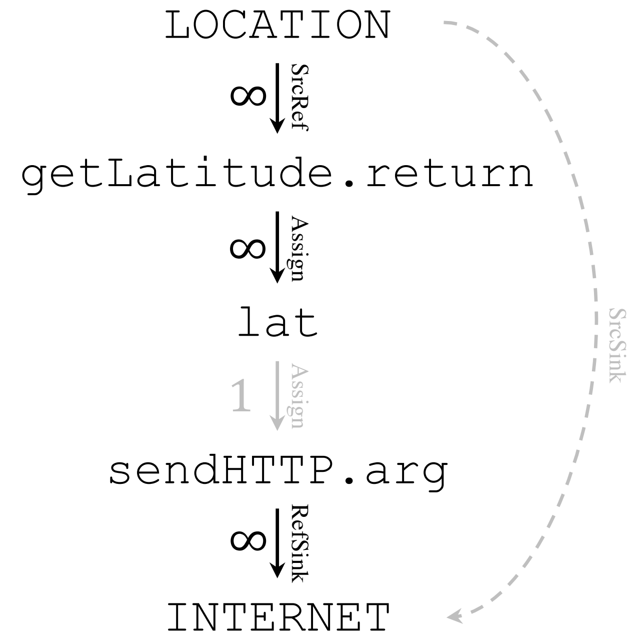
Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



Finding Valid Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { throw new Error(); sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```

Multiple Cuts

Multiple Cuts

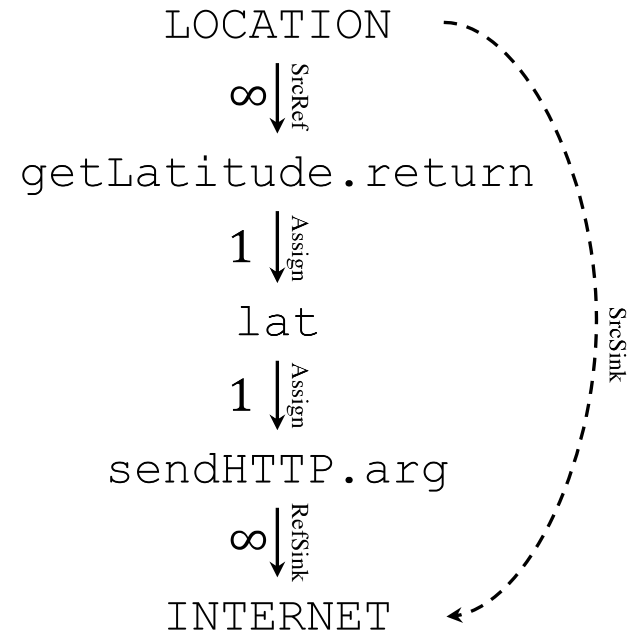
- What if a developer mistakenly answers “yes”?
  - Might delete important code!

Multiple Cuts

- What if a developer mistakenly answers “yes”?
  - Might delete important code!
- Solution: **Multiple independent** cuts
  - Developer only needs to be right once!

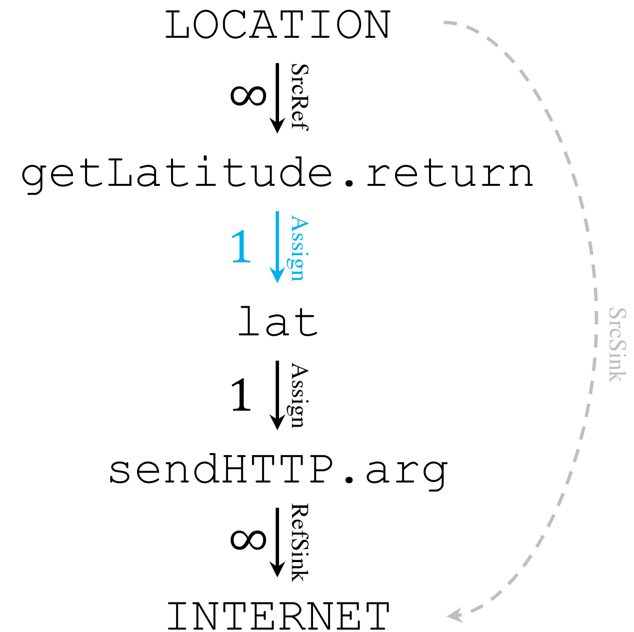
Multiple Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.       void run() {}
9.    };
10. runBenign.run();
```



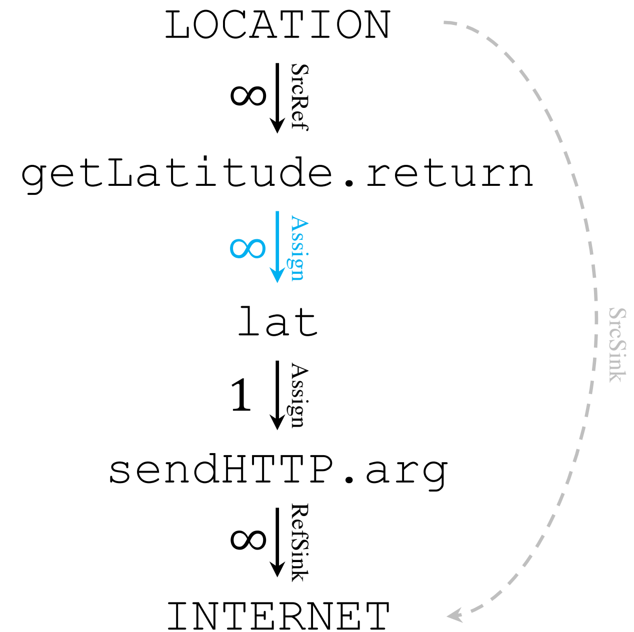
Multiple Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



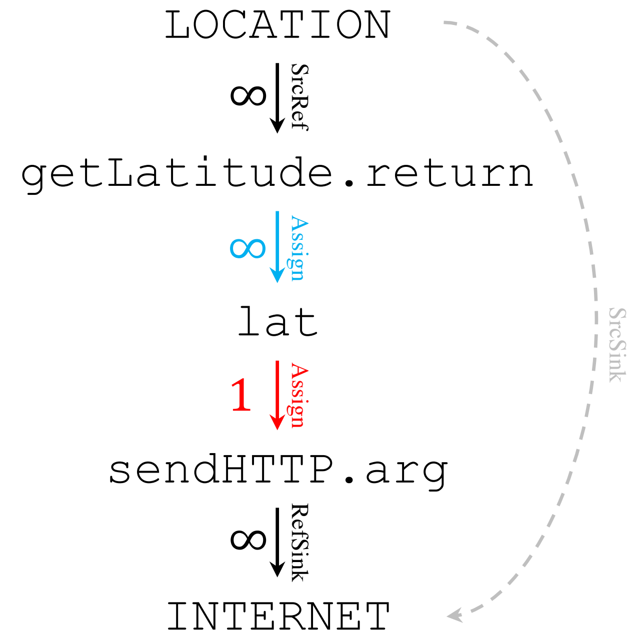
Multiple Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



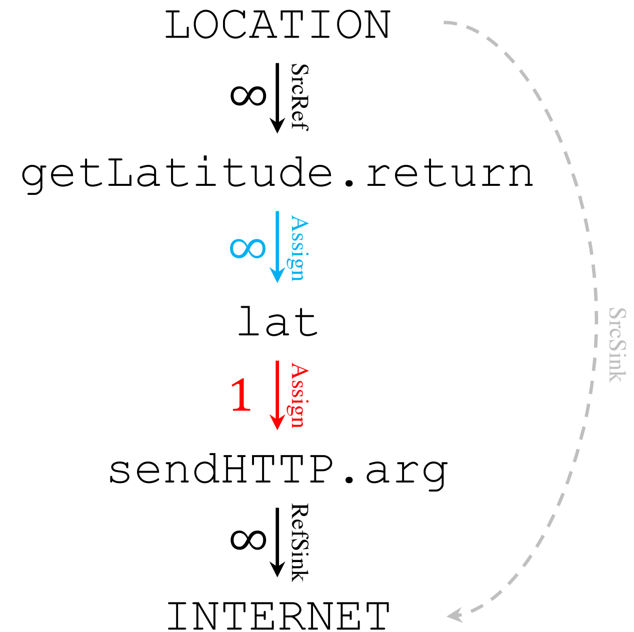
Multiple Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



Multiple Cuts

```
1.  String lat = getLatitude();
2.  Runnable runMalice =
3.    new Runnable() {
4.      void run() { sendHTTP(lat); }
5.    };
6.  Runnable runBenign =
7.    new Runnable() {
8.      void run() {}
9.    };
10. runBenign.run();
```



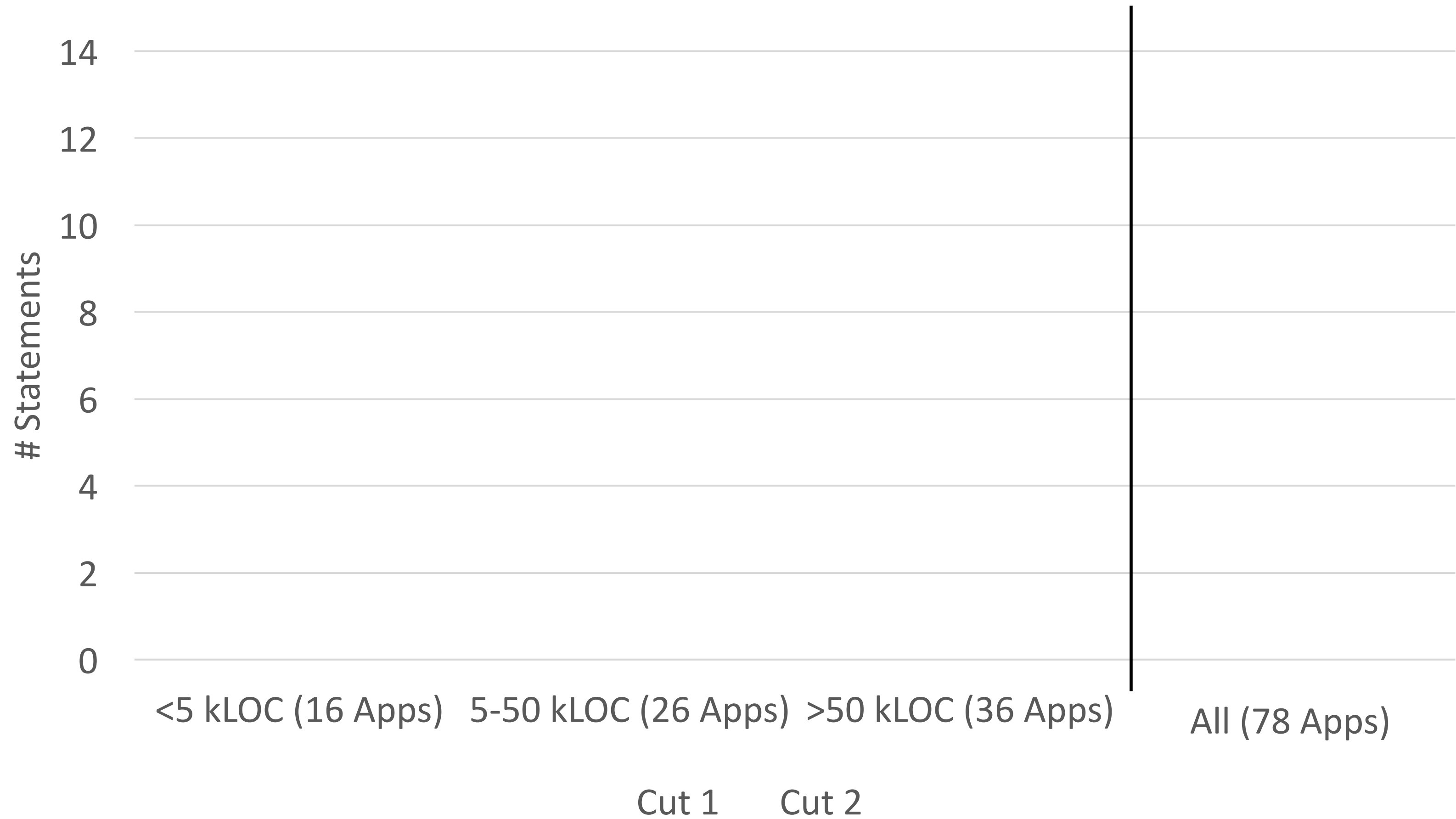
Multiple Cuts

- Program correct if ***any*** cut is valid
- Terminate only if ***every*** cut is reached

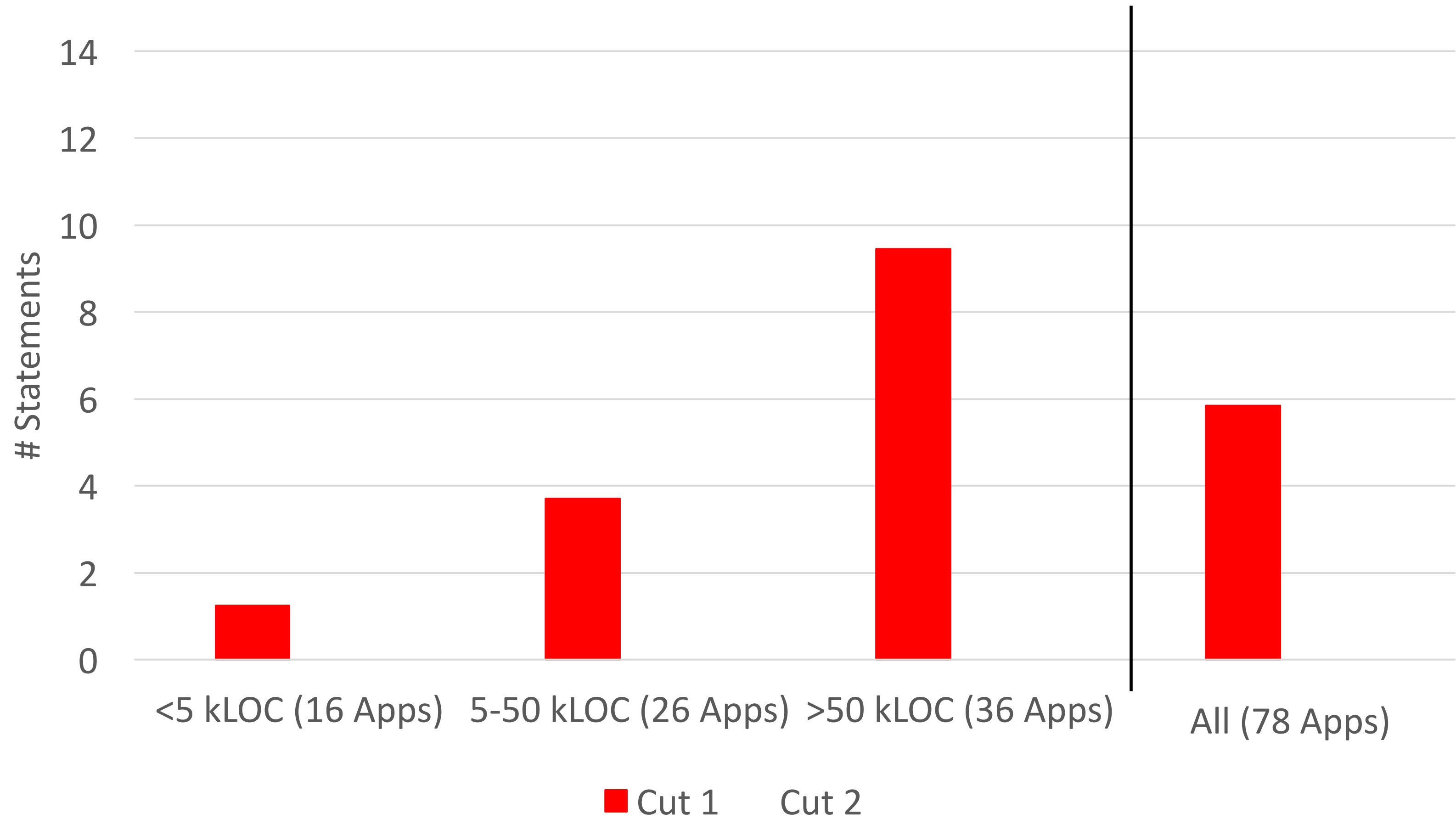
Experiments

- Ran tool on corpus of 78 Android apps
- Experiment 1: Recorded cut sizes
- Experiment 2: Interactively verified cuts

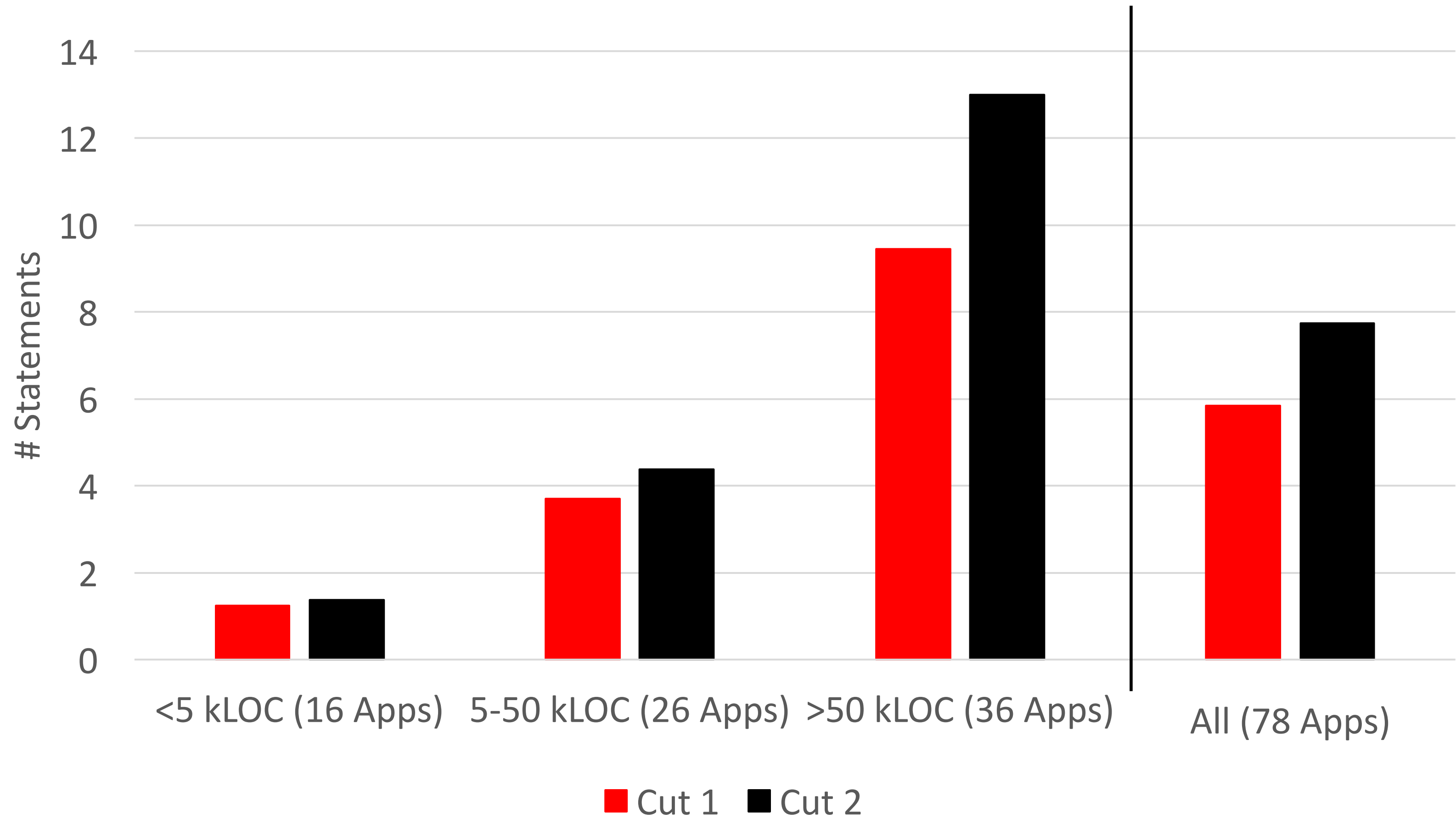
Cut sizes



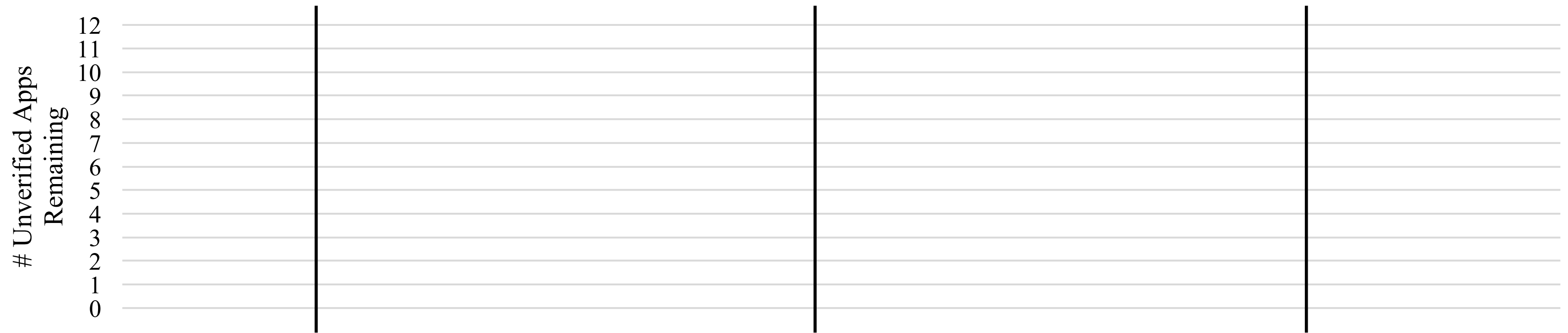
Cut sizes



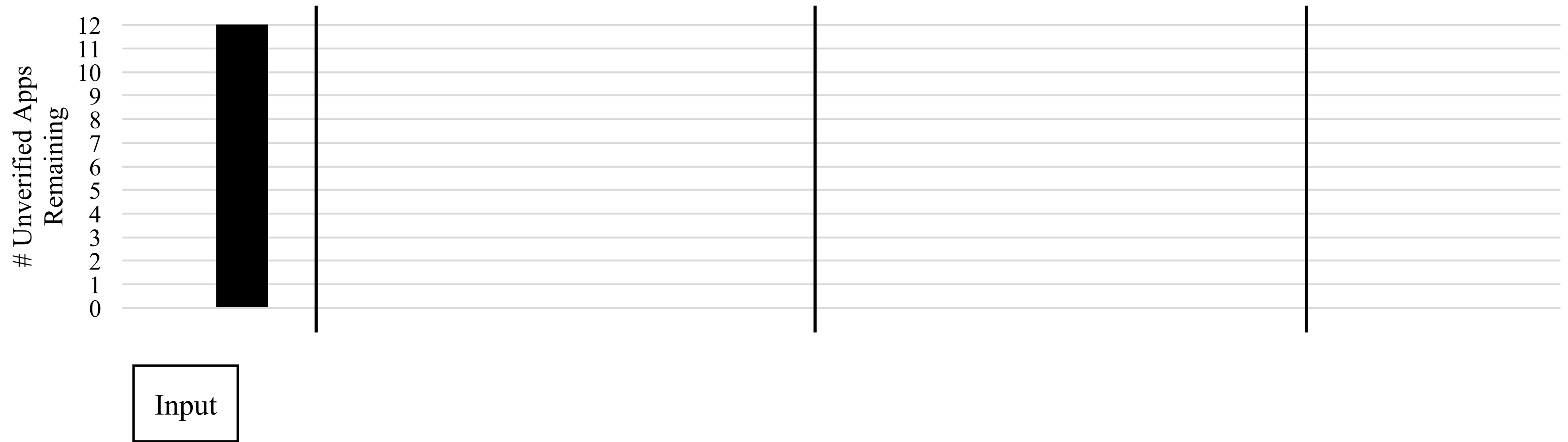
Cut sizes



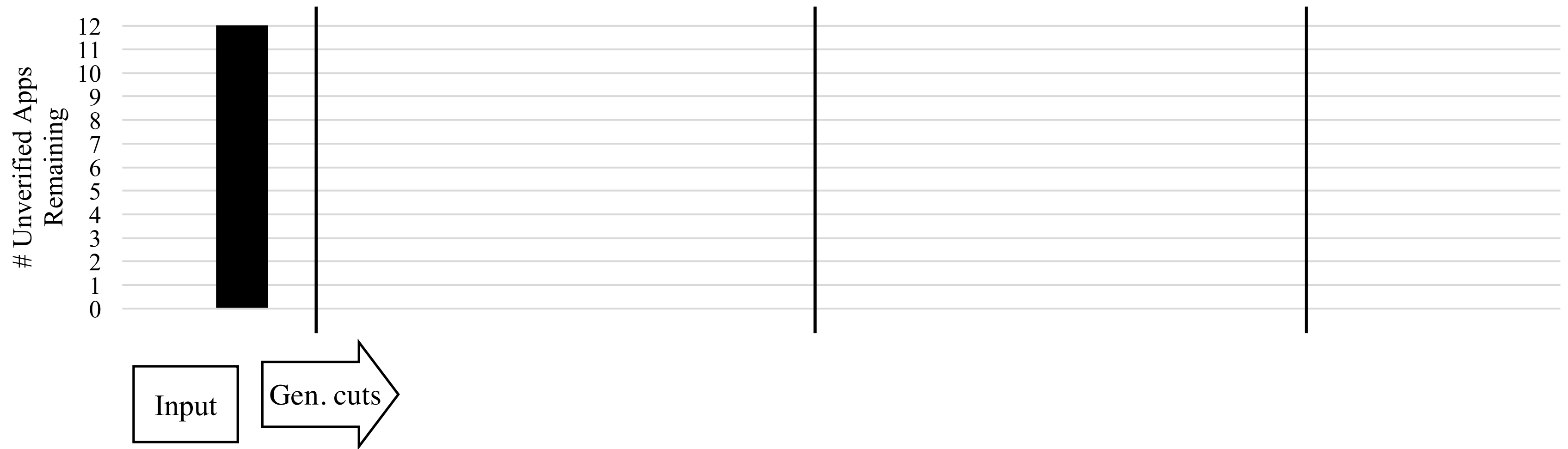
Interactive Verification



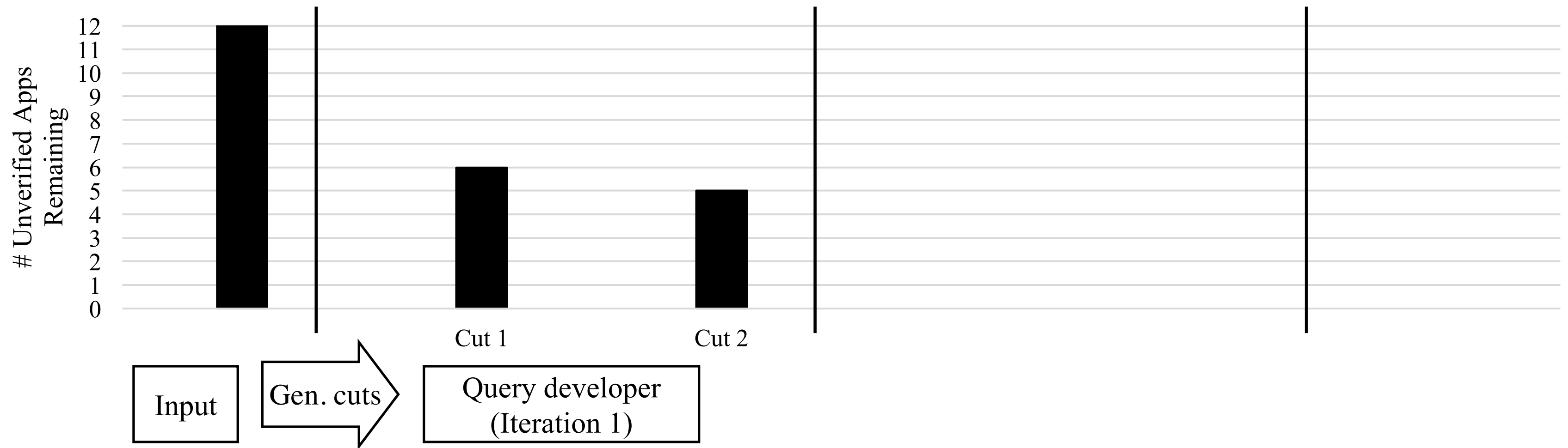
Interactive Verification



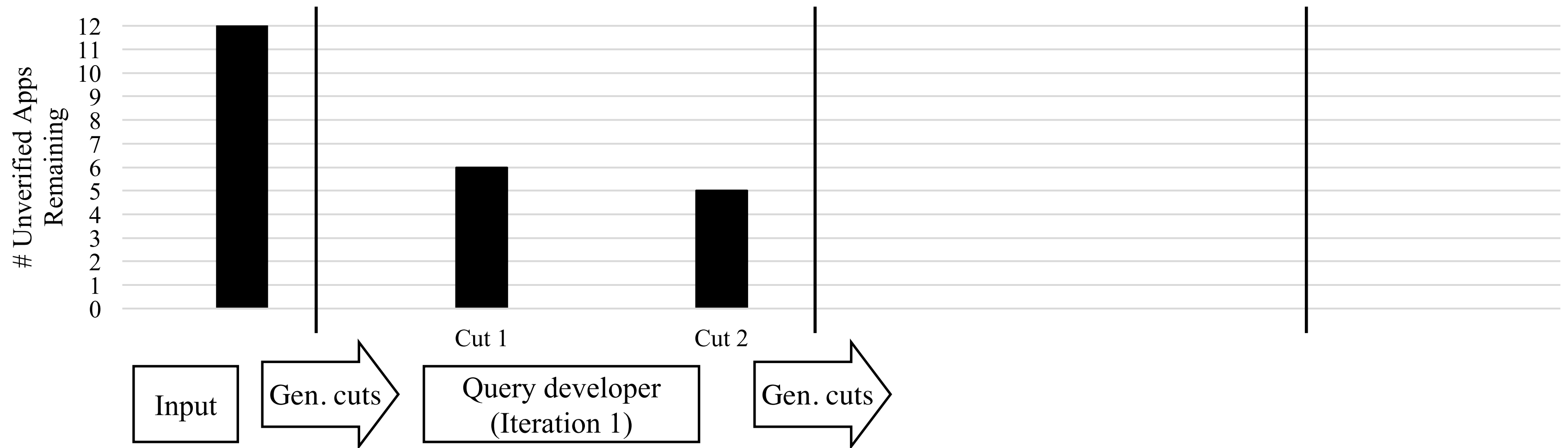
Interactive Verification



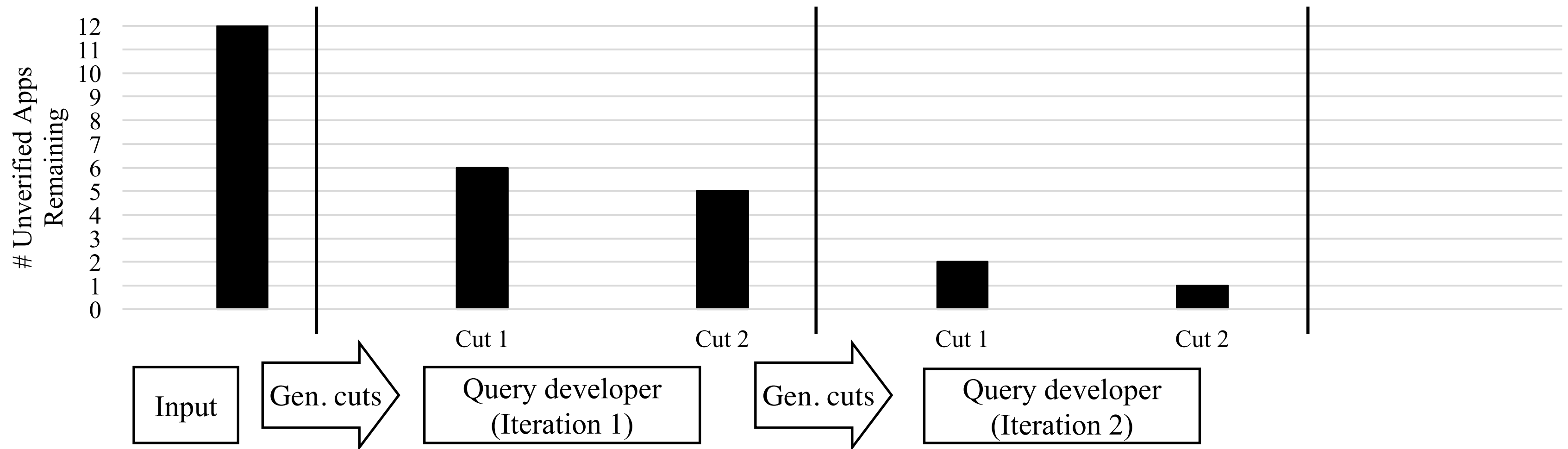
Interactive Verification



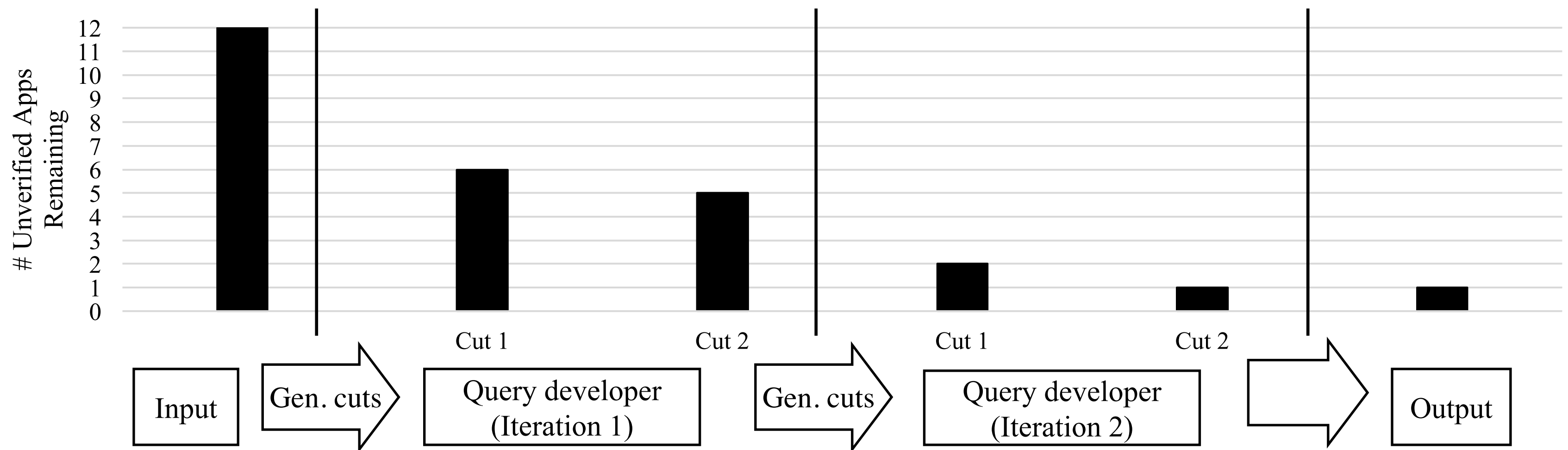
Interactive Verification



Interactive Verification



Interactive Verification



Summary

- Remove dead code with adversarial developer
  - **Step 1:** Sound static analysis
  - **Step 2:** Find cut and query developer
  - **Step 3:** Update knowledge and repeat
  - **Step 4:** *Delete* valid cut

- Experiments
  - Discharged 11 out of 12 false positives
  - Only 2 iterations needed

Conclusions

- Manual labor in “automatic” static analysis
  - Filter false positives
- A little interaction goes a long way
  - Discharged 11 out of 12 false positives due to dead code

Future Work

- Other sources of false positives
  - Reflective method calls
  - Implicit flows

- More complex security policies

References

- S. Arzt, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In PLDI, 2014.
- Y. Feng, S. Anand, I. Dillig, A. Aiken. Apposcopy: semantics based detection of Android malware through static analysis. In FSE, 2014.
- M. D. Ernst et al. Collaborative verification of information flow for a high-assurance app store. In CCS, 2014.
- I. Dillig, T. Dillig, A. Aiken. Automated error diagnosis using abductive inference. In PLDI, 2012.
- H. Zhu, T. Dillig, I. Dillig. Automated inference of library specifications for source-sink property verification. In APLAS, 2013.

Thanks!