Interactively Verifying Absence of Explicit Information Flows in Android Apps

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

App stores are increasingly the preferred mechanism for distributing software, including mobile apps (Google Play), desktop apps (Mac App Store and Ubuntu Software Center), computer games (the Steam Store), and browser extensions (Chrome Web Store). The centralized nature of these stores has important implications for security. While app stores have unprecedented ability to audit apps, users now trust hosted apps, making them more vulnerable to malware that evades detection and finds its way onto the app store. Sound static explicit information flow analysis has the potential to significantly aid human auditors, but it is handicapped by high false positive rates. Instead, auditors currently rely on a combination of dynamic analysis (which is unsound) and lightweight static analysis (which cannot identify information flows) to help detect malicious behaviors.

We propose a process for producing apps certified to be free of malicious explicit information flows. In practice, imprecision in the reachability analysis is a major source of false positive information flows that are difficult to understand and discharge. In our approach, the developer provides tests that specify what code is reachable, allowing the static analysis to restrict its search to tested code. The app hosted on the store is instrumented to enforce the provided specification (i.e., executing untested code terminates the app). We use abductive inference to minimize the necessary instrumentation, and then interact with the developer to ensure that the instrumentation only cuts unreachable code. We demonstrate the effectiveness of our approach in verifying a corpus of 77 Android apps—our interactive verification process successfully discharges 11 out of the 12 false positives.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords interactive verification; abductive inference; specifications from tests

1. Introduction

Android malware has become increasingly problematic as the popularity of the platform has skyrocketed in the past few years [67]. App stores currently identify malware using a two-step process: first, they use an automated *malware detection pipeline* to flag suspicious apps, and then a human auditor manually reviews flagged apps. The detection pipeline typically combines dynamic analysis (e.g., dynamic information flow [20]) and static analysis (e.g., identifying execution of dynamically loaded code [27]); subsequent manual examination is necessary both because the static analysis may be imprecise and because determining if suspicious behavior is malicious requires understanding the behavior's purpose (e.g., a map app should not send location data over SMS, but it may be okay for a location sharing app to do so). For example, Google uses a combination of dynamic analysis [7] and manual analysis [6] to audit Android apps on Google Play.

Important families of malware include apps that leak location, device ID, or contact information, and malware that covertly send SMS messages to premium numbers [24, 66]. Static and dynamic analyses for finding *explicit information flows* (i.e., flows arising only from data dependencies [52]) can be used to identify such malware [8, 20, 21, 24, 25]. While explicit information flows do not describe all malicious behaviors, we focus on them because they can be used to characterize a significant portion (60% according to [21]) of existing malware, especially in conjunction with additional light-weight static analyses [24, 27].

The drawback of dynamic explicit information flow analysis is the potential for false negatives. Malware can easily avoid detection by making the relevant information flow difficult to trigger. For example, the malware can detect that the execution environment is an emulator and disable malicious behaviors [61]. While static explicit information flow analysis avoids this problem, it can have a high false positive rate. Because of their global nature, false positive information flows can be time-consuming for a human auditor to understand and discharge. Furthermore, the rate at which the auditor can evaluate suspicious applications is likely the bottleneck for finding malware, implying that very few benign apps should be flagged for manual review—i.e., the malware detection pipeline must have a low false positive rate. As a consequence, app stores currently rely on approaches that achieve low false positive rates (possibly at the expense of higher false negative rates), thereby excluding static explicit information flow analysis.

Our goal is to design a sound verification process for enforcing the absence of malicious explicit information flows that guarantees no false negatives and maintains a low false positive rate. Our first key observation is that for static explicit information flow analysis, a large number of false positives are flows through unreachable code. Such false positives result from imprecision in the static reachability analysis, which can be a major problem for static analyses [9, 16]. In our setting, this imprecision is caused both by an imprecise callgraph (due to virtual method calls) and by the lack of path sensitivity. Using sound assumptions about possible entry points of an Android app can also lead to imprecision. In our experiments, 92% of false positives were flows through unreachable code. Oftentimes, the unreachable code is found in large third-party libraries used by the app.

Our second key observation is that currently the burden of identifying and discharging false positives is placed entirely on the auditor, despite the fact that the developer is most familiar with the app's code. Our approach shifts some of this burden onto the developer: we require that the developer specify which methods are reachable in the app. These reachability specifications allow the static analysis to restrict its search space to reachable code, thereby reducing the false positive rate.

In practice, we envision that the developer will provide reachability specifications by supplying tests that exercise the app code—the specification we extract is that only tested code is reachable. We use tests to avoid the senario where a developer insists (either maliciously or to avoid effort) that everything is reachable, thereby wasting auditor time and eliminating the benefits of our approach. Tests are executable, which means that the auditor can verify the correctness of the specifications. Using tests as specifications has a number of additional advantages. First, developers routinely write tests, so this approach both leverages existing tests and gives developers a familiar interface to the specification process. Second, concrete test cases can benefit the auditor in case the app must be manually examined. For our technical development, we assume that specifications are extracted from tests, though any method for obtaining correct reachability specifications suffices.

Of course, a malware developer can attempt to evade detection by specifying that the malicious code is unreachable. Our solution is simple: we enforce the developer-provided specifications by instrumenting the app to terminate if code not specified to be reachable (e.g., not covered by any of the developer-provided tests) is actually reached during runtime. The instrumented app is both consistent with the developer's specifications, and statically verified to be free of explicit information flows.

In practice, it may be difficult for developers to provide tests covering all reachable code. Therefore, we take an iterative approach to obtaining tests. To enforce the security policy, it is only necessary to terminate the app if it reaches untested code that may also lead to a malicious explicit information flow. Rather than instrument all untested program statements, we find a minimum size set of untested statements (called a *cut*) such that instrumenting these statements to terminate execution produces an app that is free of explicit information flows, and then propose this cut to the developer. If the developer finds the cut unsatisfactory, then the developer can provide new tests (or other reachability information) and repeat the process; otherwise, if the developer finds the cut satisfactory, then the cut is enforced via instrumentation as before. This process repeats until either a satisfactory cut is produced, or no satisfactory cut exists (in which case the auditor must manually review the app). We call this process *interactive verification*.

If the developer allows (accidentally or maliciously) reachable code to be instrumented, then it may be possible for the app to terminate during a benign execution. To make the process more robust against such failures, we can produce *multiple*, *disjoint* cuts. We then instrument the program to terminate only if at least one statement from every cut is reached during an execution of the app.

More formally, our goal is to infer a predicate λ (the statements in the cut) such that the security policy ϕ (lack of explicit information flows) holds provided λ holds. We compute λ using *abductive inference* [18]: given facts χ (extracted from the app $\mathcal{P}$) and security policy ϕ, abductive inference computes a minimum size predicate λ such that (i) $\chi \wedge \lambda \models \phi$ (i.e., λ together with the known program facts χ suffice to prove the security policy ϕ) and (ii) $\mathrm{SAT}(\chi \wedge \lambda)$ (i.e., λ is consistent with the known program facts χ). In our setting, we augment χ with facts extracted from the tests. Then the app $\mathcal{P}$ is instrumented to ensure that λ holds, producing a verified app $\mathcal{P}'$. Finally, we extend this process to infer multiple disjoint cuts $\lambda_1, ..., \lambda_n$, and instrument $\mathcal{P}$ to terminate only if every λ_i fails.

We propose a novel algorithm for solving the abductive inference problem for properties formulated in terms of context-free language (CFL) reachability. The security policy ϕ states that certain vertices in a graph representation of the program are unreachable. Our key insight is to formulate the CFL reachability problem as a constraint system, which we encode as an integer linear program. Finding minimum cuts in turn corresponds to a minimum solution for the integer linear program. Our work has three main contributions:

- We formalize the notion of *interactive verification* for producing verified apps using abductive inference (Section 3).
- For properties ϕ formulated in terms of CFL reachability (Section 4), we reduce the abductive inference problem to an integer linear program (Section 5).
- We implement our framework (Section 6) for producing Android apps verified to be free of explicit information flows, and show that our approach scales to large Android apps, some with hundreds of thousands of lines of bytecode (Section 7).

2. Motivating Example

Consider the Java-like code shown in Figure 1, which we call $\mathcal{P}_{\text{onCreate}}$. Suppose that a developer submits $\mathcal{P}_{\text{onCreate}}$ to the app store. The first step is to run an information flow analysis on $\mathcal{P}_{\text{onCreate}}$. We assume that the information sources and sinks are given (or inferred, see [35]). Sources and sinks are annotated in the Android framework—we inline these annotations as comments in $\mathcal{P}_{\text{onCreate}}$; the comment on line 2 says that the argument passed to `sendHTTP` is an information sink and the comment on line 6 says that the return value of `getLocation` is an information source. The goal is to prove that the user's location does not flow to the Internet:

$$\phi_{\text{flow}} = \nexists(\text{source-to-sink explicit information flow}).$$

As discussed earlier, one major source of false positive information flows is unreachable code, so we remove parts of the program that are statically proven to be unreachable. However, statically computed reachability information can be very imprecise; we focus on imprecision due to the static callgraph. For example, using a callgraph generated by class hierarchy analysis, the analysis cannot determine that line 12 cannot call `runMalice.run`. Even with a more precise callgraph, the static analysis may not be able to prove that `flag` is false in every execution. Hence, our information flow analysis finds a flow from `getLocation.return` to `sendHTTP.param`—i.e., it fails to prove ϕ_{flow}.

```
1. void leak(boolean flag, String data) {
2.   // @Sink("sendHTTP.param")
3.   if (flag) sendHTTP(data); }
4. @Entry("onCreate")
5. void onCreate() {
6.   // @Source("getLocation.return")
7.   String loc = getLocation();
8.   Runnable runMalice = new Runnable() {
9.     void run() { leak(true, loc); }}
10.  Runnable runBenign = new Runnable() {
11.     void run() { leak(false, loc); }}
12.  runBenign.run(); }
```

Figure 1. An app $\mathcal{P}_{\text{onCreate}}$ for which the static analysis potentially finds a false positive information flow. The comment in line 2 indicates that the first argument of `sendHTTP` is a sink, and the comment in line 6 indicates that the return value of `getLocation` is a source.

Our approach is to search for a *cut*, which is a subset of statements that can be removed from $\mathcal{P}_{\text{onCreate}}$ so that the resulting program $\mathcal{P}'_{\text{onCreate}}$ satisfies ϕ_{flow}. To remove a statement s from $\mathcal{P}_{\text{onCreate}}$, we terminate $\mathcal{P}_{\text{onCreate}}$ if execution reaches the program point immediately before s (i.e., if s is about to be executed). More formally, let S^* be the set of reachable program statements. Because our static analysis is sound, the set of reachable statements S computed by our static analysis overapproximates S^* (i.e., $S^* \subseteq S$). Let λ have form

$$\lambda = \bigwedge_{s \in E_\lambda} (s \notin S^*),$$

where $E_\lambda \subseteq S$. In other words, λ asserts that the subset E_λ of program statements are unreachable. Then, a cut is a predicate λ such that if every statement in $s \in E_\lambda$ is unreachable, then ϕ_{flow} holds. Any of the following choices for E_λ would allow the static analysis to prove ϕ_{flow} for $\mathcal{P}_{\text{onCreate}}$:

a. $\{$`3.sendHTTP(data)`$\}$

b. $\{$`7.getLocation()`$\}$

c. $\{$`9.leak(true,loc)`$\}$

Once our system has computed a cut λ, it returns λ for the developer to inspect, along with the instrumented program

$$\mathcal{P}'_{\text{onCreate}} = \mathcal{P}_{\text{onCreate}} - E_\lambda.$$

However, not all of the above choices for E_λ are desirable. For example, suppose our analysis infers $E_\lambda =$ (b)—then, $E_\lambda \cap S^* \neq \emptyset$, so removing E_λ from $\mathcal{P}_{\text{onCreate}}$ would result in a program that terminates during a valid execution. We call such a cut *invalid* (as opposed to a *valid* cut, which only removes unreachable statements; i.e., $E_\lambda \cap S^* = \emptyset$).

Seeing this problem, the developer returns a test that executes `onCreate`, showing that (b) is reachable. By requiring the the developer provides a test where the cut is invalid, we obtain a proof of the invalidity, which prevents the developer from automatically rejecting any cut. Upon executing `onCreate`, our system observes that (b) is reachable. Our system runs the inference algorithm to compute a new cut, this time prohibiting choice (b). The inference algorithm can return either (a) or (c). Suppose that this time, $E_\lambda =$ (a) is returned; then the developer accepts the cut E_λ because removing E_λ from $\mathcal{P}_{\text{onCreate}}$ does not remove any functionality.

Now our analysis produces $\mathcal{P}'_{\text{onCreate}} = \mathcal{P}_{\text{onCreate}} - E_\lambda$ by instrumenting $\mathcal{P}_{\text{onCreate}}$ to enforce that `3.sendHTTP(data)` is unreachable. By the definition of a cut, ϕ_{flow} provably holds for $\mathcal{P}'_{\text{onCreate}}$, so the app can be safely placed on the app store. Because (`3.sendHTTP(data)` $\notin S^*$) is true for

$\mathcal{P}_{\text{onCreate}}$, we know $\mathcal{P}'_{\text{onCreate}}$ is semantically equivalent to $\mathcal{P}_{\text{onCreate}}$. Furthermore, the instrumentation in $\mathcal{P}'_{\text{onCreate}}$ incurs no runtime overhead, since it is unreachable.

There are three alternative scenarios:

- Our information flow analysis may prove ϕ_{flow} for $\mathcal{P}_{\text{onCreate}}$, so no instrumentation is needed.
- There may not exist a valid cut, in which case the auditor must manually inspect the app.
- Finally, the developer may incorrectly accept a cut that removes reachable code (i.e., an invalid cut). In this case, the instrumented app $\mathcal{P}'_{\text{onCreate}}$ may abort during usage, but safety is maintained.

Our experiments show that a valid cut exists for the majority of false positive information flows that occur in our benchmark. Furthermore, the number of interactions required to find a valid cut is typically small, especially when using multiple cuts; see Section 7.

2.1 Analyzing Callbacks

In addition to imprecision in the callgraph, another source of imprecision is whether to treat `runMalice.run` as a *callback*. Much of an Android app's functionality is executed via callbacks that are triggered when certain system events occur, so callbacks must be annotated as program entry points. The Android framework provides thousands of callbacks; however, many of these callbacks are poorly documented, which makes manually identifying and annotating callbacks a time consuming and error prone task. If a callback annotation is missing, then reachable code may be excluded from the analysis, introducing unsoundness.

On the other hand, every callback must override an Android framework method—we call any such method a *potential callback*. Of course, not every potential callback is a true callback; for example, any method overriding `Object.equals` is a potential callback but not a true callback. In our analysis, we make the sound assumption that *every* potential callback is a callback—that is, we conservatively overestimate the set of callbacks. We then infer a cut λ as before. For example, in Figure 1, the static analysis treats `runMalice.run` as a potential callback, and thus reports the flow of location data to the Internet. The abductive inference algorithm can return the cut

$$\lambda = (\texttt{3.sendHTTP(data)} \notin S^*)$$

which as before guarantees that the program is free of explicit information flows.

While more precise analyses such as [15] exist for soundly identifying callbacks, they are still overapproximations, and furthermore may be prone to false negatives (e.g., failing to handle native code). Our approach is both simple to implement and sound.

3. Interactive Verification

We model the imprecision of static analysis by categorizing program facts as *may-facts* and *must-facts*. May-facts are facts that the static analysis cannot prove are false for all executions. For example, $(\texttt{3.sendHTTP(data)} \in S^*)$ is a may-fact for $\mathcal{P}_{\text{onCreate}}$. Conversely, must-facts χ are facts that are shown to hold for at least one concrete execution of $\mathcal{P}$. For example, since `7.getLocation()` is executed by running `onCreate`, this is a must-fact for $\mathcal{P}_{\text{onCreate}}$, i.e. $\chi = (\texttt{7.getLocation()} \in S^*) \wedge (...)$.

Our static analysis takes as input a cut λ that asserts that some may-facts are false; for example, the predicate

$$\begin{aligned}\lambda_{\texttt{9,11}} = &(\texttt{9.leak(true,loc)} \notin S^*) \\ &\wedge (\texttt{11.leak(false,loc)} \notin S^*)\end{aligned}$$

is a cut with which the static analysis can verify ϕ_{flow} for $\mathcal{P}_{\text{onCreate}}$. These assumptions have a (finite) lattice structure $(\Lambda, \leq, \top, \perp)$, where $\lambda \leq \mu$ means: if $\chi \wedge \lambda \models \phi$ holds, then $\chi \wedge \mu \models \phi$ holds as well (i.e., μ makes stronger assumptions than λ). For example, $(\texttt{9.leak(true,loc)} \notin S^*) \leq \lambda_{\texttt{9,11}}$ because $\lambda_{\texttt{9,11}}$ makes stronger assumptions. The predicate $\perp$ corresponds to no assumptions (and is guaranteed to hold), and $\top$ corresponds to assuming that all may-facts are false.

We are interested in the setting where predicates correspond to sets of program statements:

- Predicates λ correspond to sets $E_\lambda \subseteq S$:

  $$\lambda = \bigwedge_{s \in E_\lambda} (s \notin S^*),$$

  where S is the set of program statements and S^* is the set of reachable program statements. In other words, λ asserts that statements $s \in E_\lambda$ are not reachable.
- Conjunction of predicates corresponds to set union:

  $$E_{\lambda_1 \wedge \lambda_2} = E_{\lambda_1} \cup E_{\lambda_2},$$

  i.e., two cuts hold simultaneously if all of the statements from both cuts are unreachable.
- Partial order corresponds to set inclusion:

  $$\lambda \leq \mu \text{ if } E_\lambda \subseteq E_\mu.$$

  In other words, smaller sets make fewer (therefore, weaker) assumptions.
- Top and bottom: $E_\top = S$ and $E_\perp = \emptyset$.

Given $\lambda \in \Lambda$, our static analysis tries to prove $\chi \wedge \lambda \models \phi$ (i.e., it tries to prove ϕ assuming λ). When can we hope to find a valid cut λ that helps the static analysis prove ϕ? Consider three cases:

1. The static analysis proves $\chi \wedge \perp \models \phi$. Since $\perp$ always holds, the static analysis has proven that ϕ holds.

2. The static analysis cannot prove $\chi \wedge \perp \models \phi$, but proves $\chi \wedge \top \models \phi$. In this case, we can search for a valid cut $\lambda \in \Lambda$ with which the static analysis can prove ϕ.
3. The static analysis cannot prove $\chi \wedge \top \models \phi$. This means that even making best-case assumptions, the static analysis fails to prove ϕ, so no cut $\lambda \in \Lambda$ can help the static analysis prove ϕ.

In the first case, the app is already free of malicious information flows. In the third case, the app must be sent to the auditor for manual analysis. The second case is our case of interest, which we describe in more detail in the subsequent sections.

3.1 Abductive Inference

Our goal is to find a valid cut $\lambda \in \Lambda$ with which the static analysis can verify that the policy ϕ holds. Our central tool will be a variant of abductive inference where the known-facts χ are extracted from tests:

DEFINITION 3.1. Given must-facts χ extracted from dynamic executions, the *abductive inference problem* is to find a cut $\lambda \in \Lambda$ such that

$$\chi \wedge \lambda \models \phi \text{ and } \mathrm{SAT}(\chi \wedge \lambda). \tag{1}$$

Additionally, we constrain λ to be *minimal*, i.e. there does not exist $\mu \in \Lambda$ satisfying (1) such that $\mu < \lambda$.

Abductive inference essentially allows us to compute minimal specifications λ that are simultaneously consistent with the must-facts χ and verify the policy ϕ. In our setting, the abductive inference problem corresponds to finding a set E_λ such that:

- Removing E_λ from S suffices to prove ϕ_{flow} (i.e., removing E_λ from $\mathcal{P}$ guarantees that there are no source-sink flows in the resulting app $\mathcal{P}'$).
- E_λ is consistent with must-facts χ (i.e., E_λ does not contain any statements observed during a dynamic execution of $\mathcal{P}$).
- E_λ has minimum size.

We assume access to an oracle we can query to obtain the tests used to extract must-facts χ:

DEFINITION 3.2. An *oracle* $\mathcal{O}$ is a function that, on input cut λ and program $\mathcal{P}$, returns a test T_{new} showing that λ does not hold for $\mathcal{P}$, or returns $\emptyset$ if λ holds for $\mathcal{P}$.

In our setting, the oracle is the developer and the inferred cut λ is shown to the developer as the set of statements E_λ to be removed from the program. If the developer is not satisfied with E_λ, then the developer can produce a new test case such that the extracted must-facts χ_{new} satisfy $\mathrm{UNSAT}(\chi_{\text{new}} \wedge \lambda)$—i.e., χ_{new} shows that E_λ contains reachable statements, so λ is invalid. The tool updates the must-facts $\chi \leftarrow \chi \wedge \chi_{\text{new}}$ and reruns the inference procedure. This process repeats until the developer is satisfied with E_λ, upon which verification is complete. This process is performed by the refinement loop in function INTERACTIVECUT($\mathcal{P}, T$) in Algorithm 1.

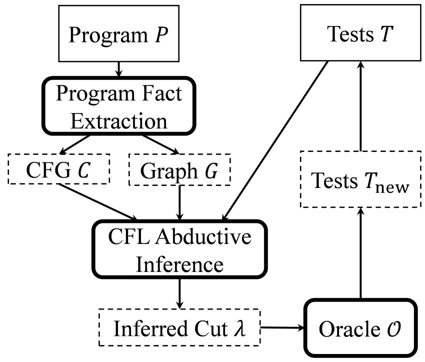

Figure 2. The interactive verification system. One iteration of the system proceeds as follows: (i) The system produces an inferred cut λ that suffices to prove absence of source-sink flows. (ii) The oracle $\mathcal{O}$ (which is the developer) either accepts λ, or generates a new test T_{new} showing that λ is invalid.

3.2 Instrumenting Cuts

Given cut λ, our framework produces

$$\mathcal{P}' \leftarrow \mathrm{INSTRUMENT}(\mathcal{P}, \lambda),$$

where $\mathcal{P}'$ is instrumented to abort if λ is violated. The instrumentation guarantees that λ holds for $\mathcal{P}'$, so ϕ holds for $\mathcal{P}'$ as well (as long as ϕ is not related to termination properties of $\mathcal{P}'$). Furthermore, if λ holds for $\mathcal{P}$, then $\mathcal{P}$ and $\mathcal{P}'$ are semantically equivalent. The procedure is summarized in Algorithm 1, and an overview of the system (for callgraph specifications discussed in Section 5) is shown in Figure 2.

The properties ϕ we have in mind are security policies, for example the policy ϕ_{flow} that no malicious explicit information flow occurs, and abductive inference computes a cut E_λ such that removing E_λ from $\mathcal{P}$ produces an app $\mathcal{P}'$ with no malicious flows. The instrumentation enforces E_λ simply by terminating execution if $s \in E_\lambda$ is reached. In our example in Figure 1, we instrument $\mathcal{P}_{\texttt{onCreate}}$ to ensure that the cut $\lambda_3 = (\texttt{3.sendHTTP(data)} \notin S^*)$ holds. Then INSTRUMENT($\mathcal{P}_{\texttt{onCreate}}, \lambda_3$) adds instrumentation that terminates $\mathcal{P}_{\texttt{onCreate}}$ if `3.sendHTTP(data)` is reached.

3.3 Improving Precision Using Multiple Cuts

We can improve precision by computing multiple sufficient cuts, which must *all* fail before the instrumentation terminates $\mathcal{P}'$. In other words, we want $\lambda_1, ..., \lambda_n$ such that

$$\chi \wedge (\lambda_1 \vee ... \vee \lambda_n) \models \phi \text{ and } \forall i,\ \mathrm{SAT}(\chi \wedge \lambda_i).$$

However, we need to avoid choosing $\lambda_1 = ... = \lambda_n$ (since then the predicates are correlated). To do so, we assume

Algorithm 1 Algorithm for interactively verifying $\mathcal{P}$. Here, the function CUT solves the abductive inference problem (algorithm described in Section 5), and the function EXTRACTFACTS constructs the must-facts χ (described in Section 4).

```
procedure INTERACTIVECUT($\mathcal{P}, \phi$)
    $T \leftarrow \emptyset$
    while true do
        $[\chi, \Lambda] \leftarrow$EXTRACTFACTS($\mathcal{P}, T$)
        $\lambda \leftarrow$CUT($\phi, \chi, \Lambda$)
        if $\lambda = \emptyset$ then
            return $\emptyset$
        end if
        $T_{\text{new}} \leftarrow \mathcal{O}(\mathcal{P}, \lambda)$
        if $T_{\text{new}} = \emptyset$ then
            return $\lambda$
        end if
        $T \leftarrow T \cup T_{\text{new}}$
    end while
end procedure
procedure INTERACTIVEVERIFY($\mathcal{P}, \phi$)
    $\lambda \leftarrow$INTERACTIVECUT($\mathcal{P}, \phi$)
    return INSTRUMENT($\mathcal{P}, \lambda$)
end procedure
```

that Λ comes with a meet operator $\sqcap$, where $\lambda \sqcap \mu$ should mean "intersection of specifications λ and μ". We require that the predicates be disjoint—i.e., $\lambda_i \sqcap \lambda_j = \perp$ for all $1 \leq i < j \leq n$. This is stronger than requiring that the predicates λ_i are distinct, but maximizes the independence of the predicates, thus making it more likely that at least one of them holds.

In our setting, where predicates λ correspond to sets $E_\lambda \subseteq S$, the meet operator is intersection:

$$E_{\lambda \sqcap \mu} = E_\lambda \cap E_\mu.$$

This satisfies the requirement $\lambda \sqcap \mu \leq \lambda$ because $E_\lambda \cap E_\mu \subseteq E_\lambda$. Now, the condition $\lambda_i \sqcap \lambda_j = \perp$ says that our cuts E_{λ_i} should be non-intersecting: $E_{\lambda_i} \cap E_{\lambda_j} = \emptyset$.

We incrementally construct the predicates λ_i. Our first predicate is $\lambda_1 \leftarrow$CUT(ϕ, χ, Λ). When computing λ_2, we need to ensure that $\lambda_1 \sqcap \lambda_2 = \perp$—i.e., we need to exclude every predicate in the *downward closure*

$$(\downarrow \{\lambda_1\}) = \{\mu \in \Lambda \mid \mu \leq \lambda_1\}$$

of λ_1 from consideration. To exclude these predicates, we add them to χ:

$$\chi_1 \leftarrow \chi \wedge \bigwedge_{\lambda \in (\downarrow \{\lambda_1\} - \{\perp\})} (\neg \lambda).$$

Now consider $\lambda_2 \leftarrow$CUT(ϕ, χ_1, Λ). Let $\nu = \lambda_1 \sqcap \lambda_2$. Note that $\nu \in (\downarrow \{\lambda_1\})$, since $\nu \leq \lambda_1$. However, CUT returns λ_2 such that SAT$(\chi_1 \wedge \lambda_2)$, and $\chi_1 = (\neg \nu) \wedge (...)$ unless $\nu \notin (\downarrow \{\lambda_1\} - \{\perp\})$, so it must be the case that $\nu = \perp$.

In general, after computing the first $i - 1$ predicates $\{\lambda_1, ..., \lambda_{i-1}\}$, we compute

$$\chi_i \leftarrow \chi_{i-1} \wedge \bigwedge_{\mu \in (\downarrow \{\lambda_{i-1}\} - \{\perp\})} (\neg \mu),$$

and choose $\lambda_i \leftarrow$CUT(ϕ, χ_i, Λ). Algorithm 2 uses this procedure to compute $\alpha = \lambda_1 \vee ... \vee \lambda_n$. Note that at some point, the problem of computing CUT becomes infeasible, after which no new sufficient cuts can be computed.

In our setting,

$$\mu \in \downarrow \{\lambda\} \text{ if } E_\mu \subseteq E_\lambda.$$

To compute multiple cuts, we need an efficient way to compute the conjunction over $(\downarrow \{\lambda_i\} - \{\perp\})$. Note that

$$\bigwedge_{\mu \in (\downarrow \{\lambda_i\} - \{\perp\})} (\neg \mu) = \bigwedge_{s \in E_{\lambda_i}} (\neg \mu_{\{s\}}) = \bigwedge_{s \in E_{\lambda_i}} (s \in S^*).$$

To see the first equality, note that the conjunction on the right-hand side is over a subset of the conjunction on the left-hand side, so the left-hand side implies the right-hand side. Conversely, every $\mu \in (\downarrow \{\lambda_i\} - \{\perp\})$ can be expressed as a (nonempty) conjunction $\mu_{\{s_1\}} \wedge ... \wedge \mu_{\{s_m\}}$, where $s_1, ..., s_m \in E_{\lambda_i}$. Therefore $\neg \mu = (\neg \mu_{\{s_1\}}) \vee ... \vee (\neg \mu_{\{s_m\}})$, which is implied by the right-hand side.

The resulting update rule is

$$\chi_i \leftarrow \chi_{i-1} \wedge \bigwedge_{s \in E_{\lambda_{i-1}}} (s \in S^*).$$

In other words, the next call to CUT assumes that every statement $s \in E_\lambda$ is in S^*. Since $\lambda_i \sqcap \lambda_j = \perp$ is equivalent to $E_{\lambda_i} \cap E_{\lambda_j} = \emptyset$, this condition is correctly enforced because χ is updated so that every statement that occurs in E_{λ_i} is prevented from occurring in E_{λ_j} (for $j > i$).

To instrument the program to enforce multiple cuts $\alpha = \lambda_1 \vee ... \vee \lambda_n$, we keep a global array of Boolean variables $[b_1, ..., b_n]$, all initialized to false. Whenever a predicate λ_i is violated, we update $b_i \leftarrow$ true. If $b_1 \wedge ... \wedge b_n$ ever becomes true, then all of the predicates λ_i have been violated and we terminate $\mathcal{P}'$.

Algorithm 2 Algorithm for computing multiple cuts.

```
procedure MULTIPLECUT($\phi, \chi, \Lambda, n$)
    $\alpha \leftarrow$ false
    for all $1 \leq k \leq n$ do
        $\lambda_i \leftarrow$CUT($\phi, \chi, \Lambda$)
        if $\lambda_i \neq \emptyset$ then
            $\alpha \leftarrow \alpha \vee \lambda_i$
            $\chi \leftarrow \chi \wedge \bigwedge_{\mu \in (\downarrow \{\lambda_i\} - \{\perp\})} (\neg \mu)$
        end if
    end for
    return $\alpha$
end procedure
```

1. `v=new X()` $\Rightarrow o \xrightarrow{\text{New}} v$
2. `u=v` $\Rightarrow v \xrightarrow{\text{Assign}} u$
3. `u.f=v` $\Rightarrow v \xrightarrow{\text{Put}[f]} u$
4. `u=v.f` $\Rightarrow v \xrightarrow{\text{Get}[f]} u$
5. `@Source(v)` $\Rightarrow v \xrightarrow{\text{SrcRef}} v$
6. `@Sink(v)` $\Rightarrow v \xrightarrow{\text{RefSink}} v$
7. $v \xrightarrow{\sigma} v' \Rightarrow v' \xrightarrow{\overline{\sigma}} v$ (where $\overline{\overline{\sigma}} = \sigma$)

Figure 3. Program fact extraction rules.

4. Background on CFL Reachability

We compute cuts for a static (explicit) information flow analysis formulated as a context-free language (CFL) reachability problem. Our analysis does not handle implicit information flows (i.e., where information is leaked due to control flow decisions), and furthermore is flow-, context-, and path-insensitive.

The first step of performing the analysis is to extract a graph $G = (V, E)$ from the given program $\mathcal{P}$, along with a context-free grammar (CFG) $C = (\Sigma, U, P, T)$, where Σ is the set of terminals, U is the set of non-terminals, P is the set of productions, and T is the start symbol. We assume that C is *normalized*, i.e. every production has the form $A \to B$ or $A \to BD$ [39]. The edges $e \in G$ are labeled with terminals $\sigma \in \Sigma$, i.e., $e = v \xrightarrow{\sigma} v'$ (for some $v, v' \in V$). The *transitive closure* G^C of G under C is the minimal sized solution to the following constraint system:

- $\dfrac{e \in G}{e \in G^C}$
- $\dfrac{v \xrightarrow{B} v' \in G^C, \quad A \to B \in C}{v \xrightarrow{A} v' \in G^C}$
- $\dfrac{v \xrightarrow{B} v'' \xrightarrow{D} v' \in G^C, \quad A \to BD \in C}{v \xrightarrow{A} v' \in G^C}$

The policy we are trying to enforce has the form $\phi = e^* \notin G^C$, where $e^* = v_{\text{source}} \xrightarrow{T} v_{\text{sink}}$ for the *source* vertex v_{source} and the *sink* vertex v_{sink}. Our exposition focuses on the case where there is a single source and a single sink, but our techniques generalize to policies ϕ excluding edges between multiple sources and sinks.

4.1 Explicit Information Flow Analysis

Given a program $\mathcal{P}$, the static analysis constructs the labeled graph $G = (V, E)$. Here, $V = \mathcal{M} \cup \mathcal{H} \cup \mathcal{V}$, where $\mathcal{M}$ is the set of methods, $\mathcal{H}$ is the set of abstract objects (which correspond to object allocation sites), and $\mathcal{V}$ is the set of reference variables. Additionally, a distinguished *source* vertex $v_{\text{source}} \in \mathcal{V}$ and a distinguished *sink* vertex $v_{\text{sink}} \in \mathcal{V}$ are annotated by the user—for example, in Figure 1, the return value of `getLocation` is annotated as a source, and the parameter of `sendHTTP` is annotated as a sink.

8. FlowsTo $\to$ New
9. FlowsTo $\to$ FlowsTo Assign
10. FlowsTo $\to$ FlowsTo Put$[f]$ $\overline{\text{FlowsTo}}$ FlowsTo Get$[f]$
11. SrcSink $\to$ SrcRef $\overline{\text{FlowsTo}}$ FlowsTo RefSink
12. $A \to A_1 ... A_k \Rightarrow \overline{A} \to \overline{A_k} ... \overline{A_1}$ (where $\overline{\overline{A}} = A$)

Figure 4. Productions for C_{flow}.

The edges $e \in E$ encode relations between the variables and abstract objects, where the relation is explained by the label $\sigma \in \Sigma$. Rules for generating G are shown in Figure 3. Rules 1-4 handle statements that are used in the points-to analysis [56]. Rule 1 says that the contents of abstract object o flow to the reference v (more formally, v may *point to* o). Rule 2 encodes the flow when a reference variable v is assigned to another reference variable u. Rules 3 and 4 record the flows induced by field reads (or *gets*) and writes (or *puts*)—note that there is a distinct put/get terminal for each field $f \in \mathcal{F}$ (where $\mathcal{F}$ is the set of fields in the program). Additionally, we handle interprocedural information flow by including (i) assignment edges from the arguments in the caller to the formal parameters of the callee, and (ii) assignments from the formal return values of the callee to the left-hand side of the invocation statement in the caller.

Rules 5-6 mark vertices as sources and sinks based on annotations in the library—Rule 5 adds a self-loop on the source vertex, and Rule 6 adds a self-loop on the sink vertex. Finally, Rule 7 is a technical device that allows us to express paths with "backwards" edges. It introduces a label $\overline{\sigma}$ to represent the reversal of an edge labeled σ. Figure 5 shows the part of the graph extracted from the code in Figure 1 (using the rules in Figure 3) that is relevant to finding the source-sink flow from `getLocation.return` to `sendHTTP.param`.

Information flows through the graph correspond to source-sink paths in the CFG C_{flow} defined as follows:

$$\begin{aligned}\Sigma_{\text{flow}} = &\{\text{New}, \text{Assign}, \text{SrcRef}, \text{RefSink}\} \\ &\cup \{\text{Put}[f], \text{Get}[f] \mid f \in \mathcal{F}\} \\ U_{\text{flow}} = &\{\text{New}, \text{Assign}, \text{FlowsTo}, \text{SrcSink}\}\end{aligned}$$

where $\mathcal{F}$ is the set of fields used in the program. We also include symbols $\overline{\sigma}$ and $\overline{A}$ in Σ_{flow} and U_{flow}, respectively. The start symbol of the grammar is $T_{\text{flow}} = \text{SrcSink}$. Productions are shown in Figure 4. Rules 8-10 build the points-to relation $o \xrightarrow{\text{FlowsTo}} v$, which means that variable $v \in \mathcal{V}$ may point to abstract object $o \in \mathcal{H}$. Rule 11 produces a source-sink edge if the value in source vertex v_{source} is aliased with the value in sink vertex v_{sink}. Finally, Rule 12 introduces a reversed edge $v' \xrightarrow{\overline{A}} v$ for every non-terminal edge $v \xrightarrow{A} v'$. In this way, Rule 12 plays the same role for non-terminal edges that Rule 10 plays for terminal edges. It is also possible to capture implicit flows through framework methods, see [11].

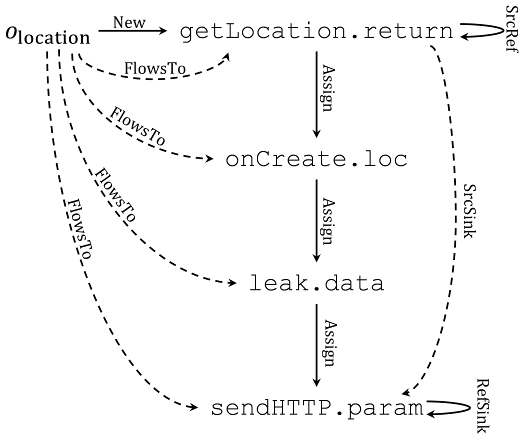

Figure 5. A part of the graph G for the code in Figure 1. Solid edges are edges extracted using the rules in Figure 1. Dashed edges are edges added by the rules in Figure 4. Backwards edges are omitted for clarity.

In Figure 5, the dashed edges are edges added when computing the transitive closure. For example, because we have edge $\texttt{return} \xrightarrow{\overline{\text{New}}} o_{\text{location}}$, Rules 8 and 12 add the edge $\texttt{return} \xrightarrow{\overline{\text{FlowsTo}}} o_{\text{location}}$. Also, because we have path

$$o_{\text{location}} \xrightarrow{\text{New}} \texttt{return} \xrightarrow{\text{Assign}} \texttt{loc} \xrightarrow{\text{Assign}} \texttt{data} \xrightarrow{\text{Assign}} \texttt{text},$$

Rules 8 and 9 add edge $o_{\text{location}} \xrightarrow{\text{FlowsTo}} \texttt{text}$. Now we have path

$$\texttt{return} \xrightarrow{\text{SrcRef}} \texttt{return} \xrightarrow{\overline{\text{FlowsTo}}} o_{\text{location}} \\ \xrightarrow{\text{FlowsTo}} \texttt{text} \xrightarrow{\text{RefSink}} \texttt{text},$$

from which Rule 11 adds $\texttt{return} \xrightarrow{\text{SrcSink}} \texttt{text}$.

5. Cuts for CFL Reachability

In this section, we describe an algorithm for performing interactive verification in the case of context-free language reachability. Let C be a context-free grammar, and let $G = (V, E)$ be a labeled graph constructed from a program $\mathcal{P}$. We consider policies ϕ of the form $\phi = (e^* \notin G^C)$, where $e^* = v_{\text{source}} \xrightarrow{T} v_{\text{sink}}$. This question can be answered in polynomial time [39]. However, the graph constructed by the static analysis in general is an approximation of the true graph $G^* = (V^*, E^*)$, i.e. $G^* \subseteq G$, which potentially introduces false positive source-sink paths.

In Section 3, we discuss predicates $E_\lambda \subseteq S$ corresponding to sets of program statements. In this section, we slightly modify notation and consider predicates $E_\lambda \subseteq E$ that correspond to sets of edges. All of our cuts are eventually converted to sets of statements—as can be seen in Figure 3, edges correspond directly to statements, except for edges of the form $v \xrightarrow{\text{SrcRef}} v$ and $v \xrightarrow{\text{RefSink}} v$ that do not occur in our cuts.

The must-facts are predicates $(e \in G^*)$ where e is an edge certain to be in G^*, whereas the may-facts are predicates $(e \in G^*)$ where it is uncertain whether $e \in G^*$. Let $E_p \subseteq E$ be this set of *may-edges*; then $e \in E_p$ corresponds to may-fact $(e \in G^*)$. Our goal is to infer specifications of the form

$$\lambda = \bigwedge_{e \in E_\lambda} (e \notin G^*)$$

where $E_\lambda \subseteq E_p$. In other words, λ specifies that the edges in E_λ are *not* in G^*. The partial ordering on the lattice Λ of specifications is $\lambda \leq \mu$ if $E_\lambda \subseteq E_\mu$; i.e., λ makes fewer assumptions about which edges are not in G^*.

In this setting, the abductive inference problem is to find a minimal subset $E_\lambda \subseteq E_p$ such that $\lambda \models \phi$ holds for $\mathcal{P}$. All else being equal, we prefer to find the smallest cuts possible, so we add the stronger constraint that $|E_\lambda|$ is minimized.

DEFINITION 5.1. Let $G_\lambda = (V, E - E_\lambda)$; i.e. the subgraph of G with edges $e \in E_\lambda$ removed. A predicate $\lambda \in \Lambda$ is a *sufficient cut* if and only if $e^* \notin G_\lambda^C$. The *CFL reachability minimum cut problem* is to find a sufficient cut λ that minimizes $|E_\lambda|$—i.e., there does not exist any sufficient cut μ such that $|E_\mu| < |E_\lambda|$.

The CFL minimum cut problem is NP-hard—we give a proof in Appendix A.

5.1 Algorithms for CFL Reachability Cuts

We describe a reduction of the minimum cut problem to an integer linear program (ILP). The objective of the ILP is to minimize $|E_\lambda|$ over the set of predicates $\{\lambda \in \Lambda \mid e^* \notin G_\lambda^C\}$. We need to translate the constraints on λ into linear inequalities. To do so, we first recast the problem by introducing the Boolean variables $\delta_e = (e \notin G_\lambda^C) \in \{0, 1\}$ for every edge $e \in G^C$—i.e., $\delta_e = 1$ if removing E_λ from E causes e to be removed from G. We can recover E_λ given the values δ_e, i.e. $E_\lambda = \{e \in E_p \mid \delta_e = 1\}$. In this formulation, the objective is to minimize $|E_\lambda| = \sum_{e \in E_p} \delta_e$.

Recall that $e = v \xrightarrow{A} v' \in G_\lambda^C$ if there exists $e' = v \xrightarrow{B} v''$ and $e'' = v'' \xrightarrow{D} v'$ in G_λ^C such that $A \to BD \in C$ (we describe the case of binary productions—the case of unary productions is similar); we denote such a triple as $e \to e'e''$. Then $\delta_e \Rightarrow (\delta_{e'} \vee \delta_{e''})$ must hold—i.e., e is removed from G_λ^C only if either e' or e'' is removed from G_λ^C. Next, for the source-sink edge e^*, we add constraint $\delta_{e^*} = 1$, which enforces $(e^* \notin G_\lambda^C)$. Finally, we require that $\delta_e = 0$ for edges $e \in E - E_p$, since these edges cannot be removed from the graph. These constraints translate into linear inequalities:

1. Productions: $\delta_e \leq \delta_{e_1} + ... + \delta_{e_k}$ for every production $e \to e_1 ... e_k$ ($k \in \{1, 2\}$).
2. Remove the source-sink edge: $\delta_{e^*} = 1$.

3. Retain must-edges: $\delta_e = 0$ for every $e \in E - E_p$

The first set of constraints follows because $\delta_e = 1$ only if $\delta_{e_i} = 1$ for some $1 \leq i \leq k$.

The number of constraints generated by this approach is intractable for the typical ILP solver, so we introduce two optimizations to reduce the number of constraints. First, we construct the constraints in a top-down manner—i.e., we only include productions contained in some derivation of e^*. If an edge $e \in G^C$ is not contained in any derivation of e^*, then the presence of e in G^C_λ does not affect the presence of e^* in G^C_λ, so e can be ignored. This optimization is implemented by first processing all productions $e^* \to e_1...e_k$ ($k \in \{1, 2\}$); for every input e_i, we recursively add productions for e_i, which recursively adds every production in some derivation of e^*.

Second, any facts added to G^C produced from only the must-edges (i.e., from edges $e \in E - E_p$) are present in G^C_λ for every $\lambda \in \Lambda$. Note that the graph $G_\top = (V, E - E_p)$ contains no edges $e \in E_p$, so the edges $e \in G^C_\top$ are produced by must-facts alone. This means that we can first compute $G^C_\top$, and then only include variables δ_e for $e \in (G^C - G^C_\top)$. More precisely, consider a production $e \to e'e''$:

1. If $e', e'' \in G^C_\top$, then $e \in G^C_\top$, so we do not add any constraints.
2. If $e' \in G^C_\top$ but $e'' \notin G^C_\top$, then we treat this as the *unary* production $e \to e''$.
3. If $e', e'' \notin G^C_\top$, then we treat this as $e \to e'e''$ as before.

Algorithm 3 summarizes the procedure. The above discussion shows that the set E_p returned by Algorithm 3 solves CFL reachability minimum cut problem.

In practice, we include one additional constraint. For $\sigma \in \Sigma$, edges $e_1 = v \xrightarrow{\sigma} v'$ and $e_2 = v' \xrightarrow{\bar{\sigma}} v$ are distinct edges, but they are derived from the same program fact. To account for this, we impose the additional constraint $\delta_{e_1} = \delta_{e_2}$ for such pairs of edges.

For example, consider the graph in Figure 5. The graph shown in Figure 6 summarizes the possible derivations of the edge $\texttt{return} \xrightarrow{\text{SrcSink}} \texttt{param}$ from the terminal edges; to distinguish this graph from G^C we refer to the edges of this graph as *arrows* and the vertices as *nodes*. There are two types of nodes—nodes corresponding to *productions* $e \to e_1...e_k$ (shown as black circles), and nodes corresponding to edges in G^C (shown as boxes containing the corresponding edge). Each production $e \to e_1...e_k$ has one incoming arrow from e, and one outgoing arrow to each of the edges $e_1, ..., e_k$. Let

$$\begin{aligned} E_p =& \{v \xrightarrow{\text{Assign}} v' \mid v \text{ formal return value}\} \\ & \cup \{v \xrightarrow{\text{Assign}} v' \mid v' \text{ formal parameter}\}. \end{aligned}$$

In other words, E_p is the set of edges corresponding to method invocations (recall that we treat each method invocation x=foo(y) as an assignment from argument y to formal parameter foo.param and an assignment from formal return value foo.return to the defined variable x).

Algorithm 3 This algorithm solves the CFL reachability minimum cut problem. Here, $\mathcal{S}$ maps variables δ_e to their value in the solution to the ILP.

```
procedure CFLCUT($C, G, E_p$)
    $G^C \leftarrow$CLOSURE($C, G$); $G^C_\top \leftarrow$CLOSURE($C, G - E_p$)
    $\mathcal{C} \leftarrow \{\delta_{e^*} = 1\}$
    $W \leftarrow [e^*]$; $X \leftarrow \{e^*\}$
    while $\neg W$.EMPTY() do
        $e \leftarrow W$.POP()
        for all $e \to e_1...e_k$ do
            $F \leftarrow \{e_i \mid e_i \notin G^C_\top\}$
            $\mathcal{C} \leftarrow \mathcal{C} \cup \{\delta_e \leq \sum_{e \in F} \delta_e\}$
            $W \leftarrow W$.CONCAT($[e \in F \mid e \notin X]$)
            $X \leftarrow X \cup \{e \in F \mid e \notin X\}$
        end for
    end while
    $\mathcal{S} \leftarrow$SOLVEILP($\min \sum_{e \in E_p} \delta_e, \mathcal{C}$)
    return $\{e \in E_p \mid \mathcal{S}(\delta_e) = 1\}$
end procedure
```

Each production generates one constraint $\delta_e \leq \delta_{e_1} + ... + \delta_{e_k}$ in the ILP, though these constraints are simplified using the two optimizations described above. Figure 7 shows the constraints generated by Algorithm 3. Constraint 1 enforces that the SrcSink edge is in the cut. Constraint 2 enforces the production

$$\begin{aligned} (\texttt{return} \xrightarrow{\text{SrcSink}} \texttt{param}) \to (& \texttt{return} \xrightarrow{\text{SrcRef}} \texttt{return} \\ & \xrightarrow{\overline{\text{FlowsTo}}} o_{\text{location}} \xrightarrow{\text{FlowsTo}} \texttt{param} \xrightarrow{\text{RefSink}} \texttt{param}) \end{aligned}$$

but the first, second, and fourth edges on the right-hand side of the production are in $G^C_\top$, so they are not included in the constraint. The third edge $o_{\text{location}} \xrightarrow{\text{FlowsTo}} \texttt{param}$ is produced from the three edges

$$\texttt{return} \xrightarrow{\text{Assign}} \texttt{loc} \xrightarrow{\text{Assign}} \texttt{data} \xrightarrow{\text{Assign}} \texttt{param},$$

which is captured by Contraints 3-5.

6. Implementation

We have implemented the interactive verification algorithm for explicit information flow analysis of Android apps within the Chord program analysis framework [40] modified to use Soot as a front end [60]. We use the ILP solver SCIP [2]. While our information flow analysis (described in Section 4) is not context-sensitive, we compute a 2-CFA points-to analysis in BDDBDDB [62] and use it to filter the points-to set we compute. More precisely, during the computation of the transitive closure of G, whenever an edge $e = o \xrightarrow{\text{FlowsTo}} v$ is produced, we check if the 2-CFA points-to set contains e. If not, we remove e from the graph and continue the computation.

App	LOC	Malware	FP/TP	E_p Choice	$\|\{e^*\}\|$	$\|E_p\|$	$\|\mathcal{V}\|$	$\|\mathcal{C}\|$	$\|\mathcal{C}_{\text{opt}}\|$	$\frac{\|\mathcal{C}_{\text{opt}}\|}{\|\mathcal{C}\|}$	Run Time (s)	$\|E_{\lambda_1}\|$	$\|E_{\lambda_2}\|$
411524	389K	Yes	TP	$E_p^{\text{param+ret}}$	4	28K	144K	1982K	264K	0.13	64.350	6	7
0C2B78	322K	Yes	TP	E_p^{param}	3	23K	491K	5076K	956K	0.19	29.483	8	10
f7d928	258K	Yes	TP	E_p^{param}	4	48K	882K	18969K	1683K	0.089	663.237	11	25
tingshu	240K	Yes	TP	E_p^{param}	5	27K	655K	7530K	1280K	0.17	101.259	26	36
16677	200K	Yes	TP	$E_p^{\text{param+ret}}$	4	40K	423K	6896K	809K	0.12	243.154	4	5
phone	198K	Yes	TP	E_p^{param}	3	8K	83K	968K	156K	0.16	11.882	11	11
583cc9	195K	Yes	TP	E_p^{param}	4	45K	792K	12575K	1526K	0.12	276.256	20	23
da8c48	190K	Yes	TP	E_p^{param}	1	5K	6K	166K	8K	0.051	0.224	1	1
4292c1	155K	Yes	TP	$E_p^{\text{param+ret}}$	3	40K	1258K	10155K	2453K	0.24	92.545	1	1
5127eb	142K	Yes	TP	$E_p^{\text{param+ret}}$	2	43K	289K	4579K	548K	0.12	426.992	5	5
1c2514	100K	Yes	TP	$E_p^{\text{param+ret}}$	1	28K	347K	3182K	649K	0.20	181.696	3	4
wifi	98K	Yes	TP	$E_p^{\text{param+ret}}$	3	31K	579K	6568K	1129K	0.17	281.255	8	16
browser	346K	No	FP	E_p^{param}	4	51K	669K	13718K	129K	0.094	32.079	7	19
00714C	248K	Yes	FP	E_p^{param}	4	51K	986K	16784K	1922K	0.11	118.056	21	25
highrail	247K	Yes	FP	$E_p^{\text{param+ret}}$	3	39K	587K	9310K	1130K	0.12	451.537	9	9
flow	131K	No	FP	$E_p^{\text{param+ret}}$	4	31K	409K	4759K	792K	0.17	48.796	11	12
calendar	125K	Yes	FP	$E_p^{\text{param+ret}}$	4	31K	226K	3916K	430K	0.11	13.233	5	5
19780d	87K	Yes	FP	$E_p^{\text{param+ret}}$	4	28K	244K	3742K	467K	0.13	894.685	21	22
aab740	86K	Yes	FP	$E_p^{\text{param+ret}}$	4	28K	241K	3695K	461K	0.13	305.971	21	21
9d1da3	56K	Yes	FP	$E_p^{\text{param+ret}}$	5	20K	217K	4321K	420K	0.097	16.862	4	4
018ee7	53K	Yes	FP	$E_p^{\text{param+ret}}$	3	21K	148K	1952K	268K	0.098	9.844	5	5
ca70f4	44K	Yes	FP	$E_p^{\text{param+ret}}$	3	10K	57K	456K	106K	0.23	10.402	3	4
battery	33K	Yes	FP	$E_p^{\text{param+ret}}$	3	14K	98K	1076K	185K	0.17	377.113	12	13
7d43c8	27K	Yes	FP	$E_p^{\text{param+ret}}$	4	9K	33K	368K	60K	0.16	5.599	3	4
Avg.	83K	–	–	–	2.32	12K	76K	2437K	338K	15.9	42.36	5.85	7.74

Figure 8. Statistics for some of the Android apps used in the experiments: the number of lines of Jimple bytecode (LOC), whether the app exhibited a false positive information flow (FP/TP), the number of source-sink edges $|\{e^* = v_{\text{source}} \xrightarrow{T} v_{\text{sink}} \in G^C\}|$, the number of may-edges $|E_p|$, the number of variables $|\mathcal{V}|$ in the ILP, the unoptimized number of constraints $|\mathcal{C}|$ and the optimized number of constraints $|\mathcal{C}_{\text{opt}}|$, the percentage $|\mathcal{C}_{\text{opt}}|$ compared to $|\mathcal{C}|$, the run time of the ILP solver, and the size of the first and second cuts (on the first iteration of our algorithm). Where relevant, we give statistics for the largest ILP solved for the given app. Also, we include the average values over the entire corpus of 77 apps (where E_p is taken to be E_p^{param}).

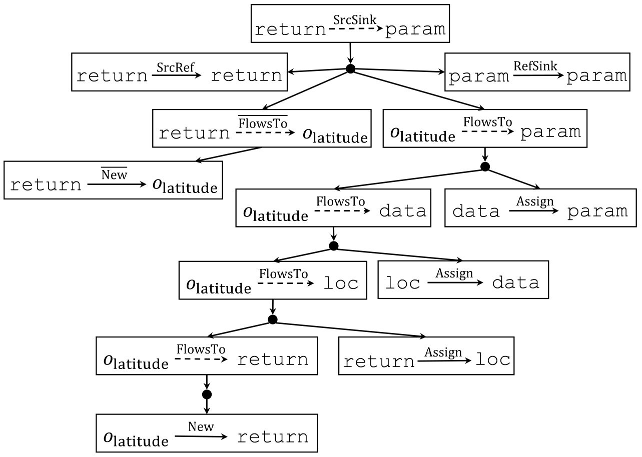

Figure 6. The derivation tree for the edge $\texttt{return} \xrightarrow{\text{SrcSink}} \texttt{param}$ in the graph in Figure 5.

$$\max \left\{ \delta(\texttt{return} \xrightarrow{\text{Assign}} \texttt{loc}) + \delta(\texttt{data} \xrightarrow{\text{Assign}} \texttt{param}) \right\}$$

subject to

1. $\delta(\texttt{return} \xrightarrow{\text{SrcSink}} \texttt{param}) = 1$
2. $\delta(\texttt{return} \xrightarrow{\text{SrcSink}} \texttt{param}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{param})$
3. $\delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{param}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{data}) + \delta(\texttt{data} \xrightarrow{\text{Assign}} \texttt{param})$
4. $\delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{data}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{loc}) + \delta(\texttt{loc} \xrightarrow{\text{Assign}} \texttt{data})$
5. $\delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{loc}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{return}) + \delta(\texttt{return} \xrightarrow{\text{Assign}} \texttt{loc})$

Figure 7. The integer linear program (ILP) corresponding to the productions shown in Figure 6.

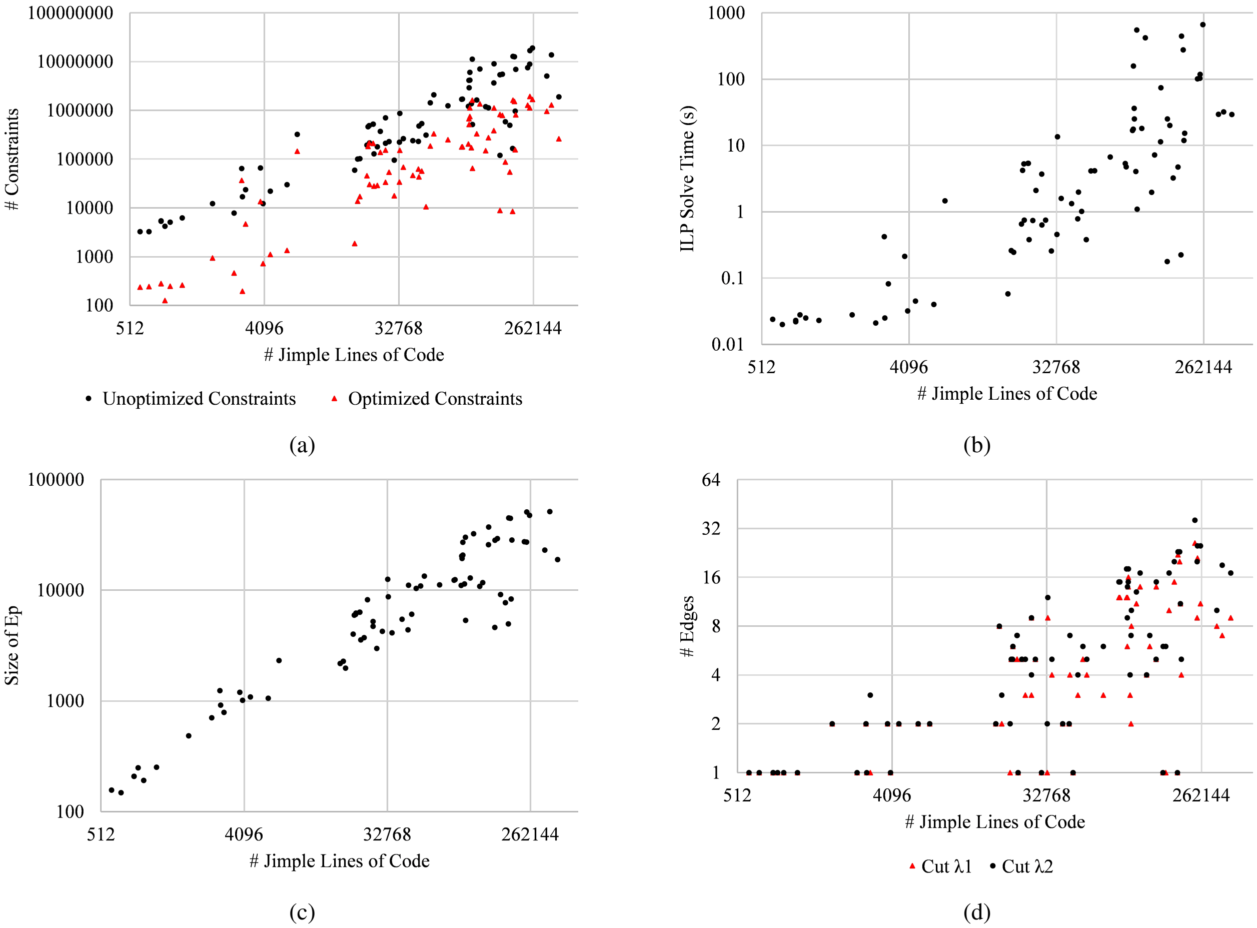

Figure 9. Statistics of the constraint system and resulting cuts for the corpus of 77 Android apps, plotted on a log-log scale: (a) number of unoptimized (black, circle) and optimized (red, triangle) constraints, (b) ILP solve time in seconds, (c) size of the search space E_p, (d) size of the first cut E_{λ_1} (red, triangle) and the second cut E_{λ_2} (black, circle).

7. Experimental Results

We demonstrate the effectiveness of our approach by interactively verifying a corpus of 77 Android apps[1], including battery monitors, games, wallpaper apps, and contact managers. These apps are a combination of malware samples and a few benign apps obtained from a major security company. The malware in this corpus contain malicious functionalities that leak sensitive information (contact data, GPS location, and the device ID) to the Internet. We have ground truth on what information is leaked for each app. Our goal is to apply Algorithm 1 to produce apps proven not to leak sensitive information. The security policy is $\phi_{\text{flow}} = v_{\text{source}} \xrightarrow{\text{SrcSink}} v_{\text{sink}} \notin G^{C_{\text{flow}}}$ (with multiple source vertices v_{source} and sink vertices v_{sink}), where C_{flow} is the context-free grammar encoding the explicit information flow analysis described in Section 4.

As described in Section 2.1, we prune the program by removing provably unreachable statements before computing information flows. Also, we make worst-case assumptions about program entry points—i.e., we assume that every potential callback is an entry point (recall that a potential callback is any method in the application that overrides a method in the Android framework).

We consider cuts consisting of method invocation statements and return statements, since these statements determine interprocedural reachability. As described in Section 5, this corresponds to choosing $E_p = E_p^{\text{param+ret}}$, where

$$\begin{aligned} E_p^{\text{param}} &= \{v \xrightarrow{\text{Assign}} v' \mid v' \text{ formal parameter}\} \\ E_p^{\text{ret}} &= \{v \xrightarrow{\text{Assign}} v' \mid v \text{ formal return value}\} \\ E_p^{\text{param+ret}} &= E_p^{\text{param}} \cup E_p^{\text{ret}}. \end{aligned}$$

The cut λ asserts that certain edges in E_p cannot happen. If an edge in E_p^{param} or E_p^{ret} is cut, then we add a statement `assert(false)` immediately before the corresponding method invocation statement.

[1] Available for download from:
`http://stanford.edu/~obastani/files/oopsla15apks.zip`

To scale to some of the largest apps in our corpus, we needed to restrict our search space of cuts—for these apps we use $E_p = E_p^{\text{param}}$ as the search space. Restricting the size of the search space can increase the size of the cuts (since the search space is strictly smaller), but in our experiments the cuts are still reasonably sized. For apps where our algorithm scaled using both E_p^{param} and $E_p^{\text{param+ret}}$, using E_p^{param} led an increase in cut size by at most a factor of about two (typically less).

In our first experiment, we run our tool on the corpus of apps and give statistics for the cuts we generate (Section 7.1). In our second experiment, we iteratively generate specifications that describe reachable code using Algorithm 1 (Section 7.2).

7.1 Inferring Cuts

We ran our tool on all the apps in our corpus. The results for twelve of the largest apps, along with all apps with false positives, are shown in Figure 8. We computed two cuts for each app, and include the sizes of each of these cuts in Figure 8. In our experience, additional cuts progressively became larger and less useful to examine (since the size of the search space reduces on every iteration), though in principle this process can safely be repeated until no new cuts can be produced—ϕ_{flow} continues to hold and having more cuts can only enlarge the set of allowed program behaviors.

We also include some statistics on the sizes of the constraint systems generated by Algorithm 3—these statistics are for the constraint system used to compute the first cut (which is the largest constraint system, though typically the size is similar for other runs). We have shown both the number of unoptimized constraints generated along with the number of constraints after applying the optimizations described in Section 5. Additionally, we include the average values over all 77 apps in the corpus (using $E_p = E_p^{\text{param}}$ for consistency).

We have plotted some of these statistics in Figure 9 for all the apps in the corpus (again, using $E_p = E_p^{\text{param}}$ for consistency). In (a), we compare the size of the unoptimized constraint system to the size after applying optimizations. As can be seen, the optimized constraint system typically reduces the size by an order of magnitude ($\approx 10\times$). The unoptimized constraint systems typically proved to be intractable for the ILP solver to optimize, but with the optimizations the solver always terminated and finished fairly quickly. We also show the running time for the ILP in (b). As can be seen, our algorithm scales well to apps with hundreds of thousands of lines of Jimple bytecode.

In (c), we show the size of E_p^{param}—this gives a sense of the size of the search space of cuts, since there are $2^{|E_p^{\text{param}}|}$ possible cuts. In (d), we show the sizes of the first two cuts produced. Most of the cuts have fewer than 16 edges, though the largest size for the first cut is 26 edges, and the largest size for the second cut is 36 edges. All of these cuts are sufficiently small so that the developer can easily verify whether the cut is valid. This suggests that the interactive verification process places little work on the developer; we further evaluate this workload in our second experiment.

7.2 Interactive Verification

In our second experiment, we manually carried out the procedure described in Algorithm 1 to produce verified apps $\mathcal{P}'$. Because the app developer is absent, we play the role of the developer. However, we are disadvantaged compared to the app developer: we only have the app bytecode, have superficial knowledge of the app's intended functionality, and lack access to the testing tools available to the developer. Furthermore, many of the apps crash when we try to run them due to incompatibilities with the Android emulator.

Thus, we provide the reachability information to our tool manually, determining which statements are reachable by reading the bytecode. The cuts are presented as a list of statements to be removed from the app, and we mark each statement as reachable or unreachable based on our inspection. For those apps that did not crash in the emulator (about half of the 12 apps) we also ran tests and found that reachability information was consistent with our specifications. In practice, we expect developers to write tests for Android apps using GUI testing frameworks such as Selendroid [53], Robotium [50], or Espresso [23].

We focused our efforts on producing cuts only for the false positives produced by our explicit information flow analysis. If the flow is a true positive, then no cut exists, so the auditor must necessarily inspect the app to determine whether it is malicious. As a consequence, in these cases little can be done to reduce the auditor workload.

The apps with false positive flows are shown in the second half of Figure 8. For each of these apps, we show the source of the false positive flow in Figure 10, and whether we determined that the cause of the false positive is due to unreachable code. These apps typically have other true positive flows—we include only sources that have false positive flows in ϕ_{flow} when performing the verification process (or else ϕ_{flow} would be false for the app).

In Figure 10, we show the results of our interactive verification process. We show two iterations of the process. For each iteration, we show the size of the cuts λ_1 and λ_2, along with the validity of each cut. The inspection of the cuts proceeded until a valid cut was found, or it was determined that no cut was possible. After just two iterations of Algorithm 1, we succeeded in producing valid cuts for all apps with false positive explicit information flows, except for the app with a false positive not due to unreachable code. This means that only two interactions with the developer were necessary. The cuts remained small after the second iteration, which shows that the entire process is feasible for the developer to carry out. In the case of the final application (browser), because the false positive was not due to unreachable code, no valid cut can be produced by our method, which means that the app would be flagged for manual review.

To demonstrate how each step of Algorithm 1 contributes to verifying each app, Figure 11 plots the number of apps remaining to be verified at each step. As can be seen, the first cut on the first iteration alone clears many of the apps (6 out of 12), and the second cut on the first iteration clears an additional app. The first cut on the second iteration clears three of the remaining apps, and the second cut clears an additional app, leaving only one app that our process failed to verify.

Whereas the auditor would initially have had to analyze all 12 false positives, our approach reduces the auditor's workload to a single false positive. In our setting, this may not seem like a huge improvement, because the auditor still needs to analyze the true positive apps. However, our corpus of apps is heavily biased towards apps with malicious behaviors. In practice, the overwhelming majority of apps received by an app store are benign, which means that even a small false positive rate leads to a huge ratio of false positives to true positives that the auditor must analyze. We achieve a 92% reduction in the number of false positives that need to be discharged by the auditor, which enables the auditor to better focus effort.

While we cannot evaluate the workload required of the developer, we describe our own experience inspecting cuts. In 10 of the cases (including the invalid cut), the cuts were very easy to evaluate, taking only a few minutes, and we are very confident of the results. The remaining 2 cases were considerably more difficult, and took up to two hours each, leaving more room for error. This difficulty was primarily a consequence of code obfuscation. We believe that it would be significantly easier for the developer, who understands the app and has source code, to examine the cuts. While most cuts were in third-party libraries, the developer has knowledge of which library features they use, which should aid them in evaluating the correctness of the cut. Furthermore, developers often maintain high-coverage test suites, which we believe would also aid the process.

We found two sources of imprecision that led to the false positives. The first was the presence of a conditional to the following effect:

```
if (hasLocationPermission()) { leakLoc(); }
```

In cases where the app did not have permissions to access location, this caused the information flow analysis to report a false positive. The second was due to our sound assumption that every potential callback is an entry point, which caused unreachable code to be marked as reachable. In both cases, our algorithm can find cuts removing the unreachable code. In the case of the app for which no cut could be found, we believe the false positive was due to insufficient context sensitivity, not flows through unreachable code.

8. Related Work

Our approach to performing iterative verification is related to the following prior techniques.

Abductive inference. Abductive inference has been applied to aid developers in understanding error reports [18], to infer program invariants [19], to infer information flow specifications for library functions [68], and for abstraction refinement [65]. We present a general approach for inferring multiple predicates, as well as optimized algorithms for abductive inference in the case where ϕ is described using CFL reachability. Also, [34] presents a related approach to guide sanitizer placement. Their approach is fully automated, but requires runtime taint tracking (though they minimize use of taint tracking).

Specifications from tests. There has been much work on extracting specifications from dynamic executions, with applications to verification [22, 54], proving equivalence of programs [55], finding good abstractions [41], and proving program termination [44]. In another line of work [26], must-facts (extracted from guided dynamic executions) have been used to avoid spending effort trying to discharge true positives. In contrast, we use must-facts extracted from tests as specifications for may-facts—i.e., we *enforce* that may-facts not observed in tests are invalid using instrumentation, and use abductive inference to minimize the amount of instrumentation required.

Dynamic instrumentation for safety. Instrumenting programs to ensure safety properties is well-studied, for example to enforce type safety [28, 42] and to ensure control-flow integrity [1]. Our work applies similar principles to ensure the integrity of information flows, which is more challenging because information flows are global properties. In [13], instrumentation is guided by testing: only reflective calls observed during execution are permitted. Their instrumentation issues warnings to the user for potential unsoundness in the static analysis. Finally, there has been work on modifying programs to coerce potentially problematic inputs into acceptable forms manually specified by the user [48, 49].

Security applications. Static information flow analysis has been applied previously to the verification of security policies [8, 21, 25, 37, 59, 63]. Our work makes static verification of security policies more practical—rather than employing a large amount of manual labor to discharge false positives, our framework allows the auditor to instrument the program $\mathcal{P}$ to enforce the security policy, with the guarantee that the instrumentation is consistent with tests given by the developer.

The approach in [21] shares our goal of moving the burden of verification to the (possibly adversarial) developer; they require the developer to annotate the source code with information types to guide the auditing process. Compared to [21], our approach does not require source code (commercial app stores typically do not have access to source

App	v_{source}	Cause	Iteration 1				Iteration 2			
			$\lvert E_{\lambda_1}\rvert$	$\lvert E_{\lambda_2}\rvert$	$\chi \wedge \lambda_1 \overset{?}{\models} \phi_{\text{flow}}$	$\chi \wedge \lambda_2 \overset{?}{\models} \phi_{\text{flow}}$	$\lvert E_{\lambda_1}\rvert$	$\lvert E_{\lambda_2}\rvert$	$\chi \wedge \lambda_1 \overset{?}{\models} \phi_{\text{flow}}$	$\chi \wedge \lambda_2 \overset{?}{\models} \phi_{\text{flow}}$
browser	location	u.k.	5	6	No	No	5	None	No	No
00714C	contacts	u.r.	2	8	Yes	-	-	-	-	-
highrail	device ID	u.r.	1	2	Yes	-	-	-	-	-
flow	contacts	u.r.	2	3	Yes	-	-	-	-	-
calendar	location	u.r.	3	3	Yes	-	-	-	-	-
19780d	contacts	u.r.	1	1	No	No	2	9	Yes	-
aab740	contacts	u.r.	1	1	No	No	2	9	Yes	-
9d1da3	location	u.r.	4	4	No	No	5	5	No	Yes
018ee7	location	u.r.	4	4	Yes	-	-	-	-	-
battery	location	u.r.	8	8	No	Yes	-	-	-	-
ca7b26	location	u.r.	4	4	Yes	-	-	-	-	-
7d43c8	location	u.r.	3	4	No	No	4	4	Yes	-

Figure 10. Size and validity of cuts generated by Algorithm 3 for apps with false positive flows. "None" means no cut could be generated. For "Cause", "u.k." means the cause is unknown, and "u.r." means the information flow is unreachable.

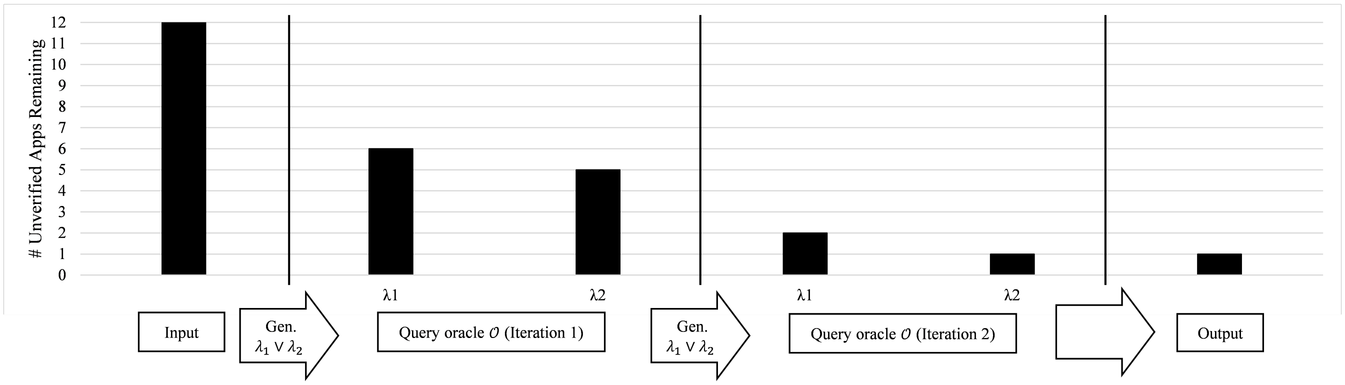

Figure 11. Visualization of how many apps are successfully verified at each step of the process. Algorithm 1 is run on each of the 12 input apps that have a false positive explicit information flow. The x-axis describes the various points in the process, and the y-axis describes the number of apps remaining to be verified at each point.

code) and leverages existing test suites to produce specifications rather than requiring annotations specific to information flow.

Dynamic taint tracking has been applied to produce programs that terminate execution upon violation of the security policy [20]. To the best of our knowledge, existing approaches to enforce information flow policies require instrumenting the entire program (or modifying the runtime environment). In contrast, our approach uses very minimal instrumentation, and often places that instrumentation in unreachable code where it will have zero runtime cost. Other approaches for restricting app behaviors have been proposed, for example [51], but the policies enforced are local (e.g., disallowing calls to certain library methods).

Specification inference. There is a large body of work on specification inference; see [11] for a survey. There has also been work specifically on inferring callback specifications [15]. Their approach, which reduces the false positive rate, complements our approach since it would help further reduce the work required of the developer.

9. Conclusions

Given a program $\mathcal{P}$ and a policy ϕ, our framework minimally instruments $\mathcal{P}$ to ensure that ϕ holds. This instrumentation is guaranteed to be consistent with given test cases, and furthermore the developer can interact with the process to produce suitable cuts. Our approach to handling false positives has the potential to make automated verification of the absence of explicit information flows a more practical approach for security auditors to produce safe and usable programs. We have applied this approach to verify the absence of malicious explicit information flows in a corpus of 77 Android apps. For 11 out of 12 false positives information flows we found, our tool produced valid cuts to enforce ϕ_{flow}.

9.1 Future Work

In our experience, Android malware to date does not rely on sophisticated techniques to hide malicious behavior. We believe this is because such malware predominantly appears on third-party app stores where sophisticated security auditing (either manual or automatic) is unavailable. Android malware is likely to become more sophisticated over time, in which case the limitations in our static analysis may be exploited. In particular, it may be interesting to study the following limitations to our current analysis:

- Implicit flows: While we do not take into account the possibility of implicit flows in the application [52], we can easily extend our technique to do so—we can include "transfer" edges in the analysis that pass taint from variables used in conditionals to variables used in branches.
- Exception analysis: Our analysis does not currently track flows due to exceptional control flow. There has been recent work on exception handling [14].
- Reflection: Our analysis cannot resolve method calls made using the Java reflection API, so we treat such calls as no-ops. There has been recent work on handling reflective method calls [13, 36].
- Missing models: Our information flow analysis depends on *information flow models* [11, 17, 68], which means that missing models can introduce unsoundness into our analysis. For the apps in our experiments, we have carefully searched for potential missing models.

For each of these settings, a key challenge is handling the high false positive rate from a sound analysis (implicit flows [29], exceptions [14], reflection [13], and missing models [11]). Our technique may therefore be particularly applicable to these settings, though the search space of cuts may need to be modified.

Acknowledgments

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.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] T. Achterberg. SCIP: solving constraint integer programs. In *Mathematical Programming Computation*, 2009.

[3] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, P. Hawkins. An overview of the Saturn project. In *PASTE*, 43-48, 2007.

[4] R. Alur, P. Černý, P. Madhusudan, W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, 2005.

[5] G. Ammons, R. Bodík, J. Larus. Mining specifications. In *POPL*, 2002.

[6] Android developers blog, Mar. 2015. `http://android-developers.blogspot.com/2015/03/creating-better-user-experiences-on.html`

[7] Android security blog, Feb. 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.

[9] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh. Using static analysis to find bugs. In *IEEE Software*, 2008.

[10] T. Ball, S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

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

[12] N. Beckman, A. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, 2011.

[13] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, M. Mezini. Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[14] Exception analysis and points-to analysis: better together. M. Bravenboer, Y. Smaragdakis. In *ISSTA*, 2009.

[15] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, Y. Chen. EdgeMiner: automatically detecting implicit control flow transitions through the Android framework. In *NDSS*, 2015.

[16] A. Chou, B. Chelf, D. Engler, M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *ASPLOS*, 2000.

[17] L. Clapp, S. Anand, A. Aiken. Modelgen: mining explicit information flow specifications from concrete executions. In *ISSTA*, 2015.

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

[19] I. Dillig, T. Dillig, B. Li, K. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, 2013.

[20] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, A. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[21] M. D. Ernst, R. Just, S. Millstein, W. M. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *CCS*, 2014.

[22] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao. The Daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2007.

[23] Espresso, Jan. 2015. `https://code.google.com/p/android-test-kit/wiki/Espresso`.

[24] Y. Feng, S. Anand, I. Dillig, A. Aiken. Apposcopy: semantics-based detection of Android malware through static analysis. In *FSE*, 2014.

[25] A. P. Fuchs, A. Chaudhuri, J. S. Foster. SCanDroid: automated security certification of Android applications. In *IEEE Symposium on Security and Privacy*, 2010.

[26] P. Godefroid, A. V. Nori, S. K. Rajamani, S. D. Tetali Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, 2010.

[27] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *MobiSys'*, 2012.

[28] F. Henglein. Global tagging optimization by type inference. In *ACM Conference on Lisp & Functional Programming*, 1992.

[29] D. King, B. Hicks, M. Hicks, T. Jaeger. Implicit flows: cant live with em, cant live without em. In *ICISS*, 2008.

[30] J. Kodumal, A. Aiken. Banshee: a scalable constraint-based analysis toolkit. In *SAS*, 2005.

[31] J. Kodumal, A. Aiken. Regularly annotated set constraints. In *PLDI*, 2007.

[32] J. Kodumal, A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, 2004.

[33] T. Kremenek, P. Twohey, G. Back, A. Ng, D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI*, 2006.

[34] B. Livshits, S. Chong. Towards fully automated placement of security sanitizers and declassifiers. In *POPL*, 2013.

[35] B. Livshits, A. V. Nori, S. K. Rajamani, A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[36] Y. Li, T. Tan, J. Xue Effective soundness-guided reflection analysis. In *SAS*, 2015.

[37] B. Livshits, M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[38] B. Livshits, M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *FSE*, 2003.

[39] D. Melski, T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. In *Theoretical Computer Science*, 248(1):29-98, 2000.

[40] M. Naik, A. Aiken, J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[41] M. Naik, H. Yang, G. Castelnuovo, M. Sagiv. Abstractions from tests. In *POPL*, 2012.

[42] G. C. Necula, J. Condit, M. Harren, S. McPeak, W. Weimer. CCured: type-safe retrofitting of legacy software. In *TOPLAS*, 2005.

[43] J. W. Nimmer, M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, 2002.

[44] A. Nori, R. Sharma. Termination proofs from tests. In *FSE*, 2013.

[45] M. K. Ramanathan, A. Grama, S. Jagannathan. Static specification inference using predicate mining. In *PLDI*, 2007.

[46] T. Reps. Program analysis via graph reachability. In *ILPS*, 1997.

[47] T. Reps, S. Horwitz, M. Sagiv. Precise interprocedural data flow analysis via graph reachability. In *POPL*, 1995.

[48] M. Rinard. Acceptability-oriented computing In *OOPSLA*, 2003.

[49] M. Rinard. Living in the comfort zone. In *OOPSLA*, 2007.

[50] Robotium, 2015. `https://code.google.com/p/robotium/`.

[51] G. Russello, A. B. Jimenez, H. Naderi, W. van der Mark. FireDroid: hardening security in almost-stock Android. In *ACSAC*, 2013.

[52] A. Sabelfeld, A. C. Myers. Language-based information-flow security. In *IEEE Journal on Selected Areas in Communications*, 2003.

[53] Selendroid, 2015. `http://selendroid.io/`.

[54] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, A. Nori. Verification as learning geometric concepts. In *SAS*, 2013.

[55] R. Sharma, E. Schkufza, B. Churchill, A. Aiken. Data-driven equivalence checking. In *OOPSLA*, 2013.

[56] M. Sridharan, D. Gopan, L. Shan, R. Bodik. Demand-driven points-to analysis for Java. In *OOPSLA*, 2005.

[57] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, R. Berg. F4F: taint analysis of framework-based web applications. In *OOPSLA*, 2011.

[58] M. Sridharan, R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

[59] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[60] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan. Soot: a Java bytecode optimization framework. In *CASCON*, 1999.

[61] T. Vidas, N. Cristin. Evading Android runtime analysis via sandbox detection. In *ASIA CCS*, 2014.

[62] J. Whaley, M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *OOPSLA*, 2004.

[63] Y. Xie, A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[64] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.

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

[66] Y. Zhou, X. Jiang. Dissecting Android malware: characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[67] Y. Zhou, Z. Wang, W. Zhou, X. Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In *NDSS*, 2012.

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

A. CFL Minimum Cut is NP-Hard

THEOREM A.1. The CFL minimum cut problem is NP-hard.

Proof: We prove the theorem by reducing the minimum vertex cover problem to the CFL minimum cut problem. Consider the minimum vertex cover problem for a given undirected graph $G = (V, E)$, where $V = \{v_1, ..., v_n\}$. We construct the following directed, labeled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and context-free grammar C such that a CFL minimum cut for $\mathcal{G}$ and C corresponds to a minimum vertex cover for G.

Context-free grammar C. The context-free grammar C is defined to be:

1. Alphabet $\Sigma = \{a, b, c, d\}$.
2. A single production $T \to abcbd$, where T is the start symbol.

Graph $\mathcal{G}$. The graph $\mathcal{G}$ is defined to be:

1. We have vertices $x^*, y^* \in \mathcal{G}$.
2. For each $i \in \{1, ..., n\}$, we have vertices $x_i, y_i \in \mathcal{G}$.
3. For each $i \in \{1, ..., n\}$, $\mathcal{E}$ contains the edges
$$x^* \xrightarrow{a} x_i \xrightarrow{b} y_i \xrightarrow{d} y^*.$$
4. For each edge $(v_i, v_j) \in E$, we have edges
$$y_i \xrightarrow{c} x_j, \qquad y_j \xrightarrow{c} x_i.$$

CFL minimum cut problem. Finally, the specification of the CFL minimum cut problem is as follows:

1. The source vertex is x^*.
2. The sink vertex is y^*.
3. The edges labeled b have weight 1.
4. All other edges have weight ∞.

CFL minimum cut $\Rightarrow$ vertex cover. First, we claim that given a cut $\mathcal{E}_{\text{cut}} = \{x_i \xrightarrow{b} y_i\}$, the corresponding vertices

$$V_{\text{cover}} = \{v_i \mid x_i \xrightarrow{b} y_i \in \mathcal{E}_{\text{cut}}\}$$

form a cover. To see this, note that for every edge $(v_i, v_j) \in E$, we have path

$$x^* \xrightarrow{a} x_i \xrightarrow{b} y_i \xrightarrow{c} x_j \xrightarrow{b} y_j \xrightarrow{d} y^*$$

in $\mathcal{E}$. By the definition of a cut, we know that

$$x_i \xrightarrow{b} y_i \in \mathcal{E}_{\text{cut}} \text{ or } x_j \xrightarrow{b} y_j \in \mathcal{E}_{\text{cut}}.$$

As a consequence, by the definition of V_{cover}, we have

$$v_i \in V_{\text{cover}} \text{ or } v_j \in V_{\text{cover}},$$

so V_{cover} is a vertex cover as claimed.

Vertex cover $\Rightarrow$ CFL minimum cut. Conversely, we claim that given a cover $V_{\text{cover}} = \{v_i\}$, the corresponding edges

$$\mathcal{E}_{\text{cut}} = \{x_i \xrightarrow{b} y_i \mid v_i \in V_{\text{cover}}\}$$

form a cut. To see this, note that for every path

$$x^* \xrightarrow{a} x_i \xrightarrow{b} y_i \xrightarrow{c} x_j \xrightarrow{b} y_j \xrightarrow{d} y^*,$$

we have $(v_i, v_j) \in E$, so by the definition of a cover, we have

$$v_i \in V_{\text{cover}} \text{ or } v_j \in V_{\text{cover}}.$$

As a consequence, by the definition of $\mathcal{E}_{\text{cut}}$, we have

$$x_i \xrightarrow{b} y_i \in \mathcal{E}_{\text{cut}} \text{ or } x_j \xrightarrow{b} y_j \in \mathcal{E}_{\text{cut}}.$$

Furthermore, every CFL source-sink path in $\mathcal{G}$ has this form, so $\mathcal{E}_{\text{cut}}$ is a cut as claimed.

Finally, note that for both directions of the proof,

$$|V_{\text{cover}}| = |\mathcal{E}_{\text{cut}}|,$$

so in particular, a minimum cut corresponds to a minimum cover (and vice versa). □