EXPLORER : Query- and Demand-Driven Exploration of Interprocedural Control Flow Properties

Yu Feng Xinyu Wang Isil Dillig Calvin Lin
University of Texas at Austin, USA
{ yufeng, xwang, isil, lin }@cs.utexas.edu

Abstract

This paper describes a general framework—and its implementation in a tool called EXPLORER–for statically answering a class of interprocedural control flow queries about Java programs. EXPLORER allows users to formulate queries about feasible callstack configurations using regular expressions, and it employs a precise, demand-driven algorithm for answering such queries. Specifically, EXPLORER constructs an automaton $\mathcal{A}$ that is iteratively refined until either the language accepted by $\mathcal{A}$ is empty (meaning that the query has been refuted) or until no further refinement is possible based on a precise, context-sensitive abstraction of the program. We evaluate EXPLORER by applying it to three different program analysis tasks, namely, (1) analysis of the observer design pattern in Java, (2) identification of a class of performance bugs, and (3) analysis of inter-component communication in Android applications. Our evaluation shows that EXPLORER is both efficient and precise.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language constructs and features—Frameworks

General Terms Algorithms, Languages, Program analysis

Keywords Demand-driven analysis, points-to analysis, query language, control flow properties

1. Introduction

Many problems in program analysis and software engineering require answers to queries about a program's interprocedural control flow properties. For instance, consider the following questions that commonly arise in software analysis and understanding tasks:

- *Is method m reachable from* `main`*?* This question is useful for identifying interprocedurally unreachable code.
- *Does an application call any method that is deemed to be a potential security risk?* Many malware detectors need to answer questions of this sort [16, 18, 41].
- *Can method m be called—either directly or transitively—from some other method m'?* This question arises as a common program analysis subtask, for example, in automated testing and analysis of Android applications [16, 28, 39]. Furthermore, programmers often ask such questions to assist them in various debugging, refactoring, or code understanding tasks.

To answer such questions, users need to construct a static callgraph of the program and then perform some kind of analysis on it (e.g., reachability). Unfortunately, there is a steep tradeoff between the precision of a callgraph and the cost of constructing it. For example, callgraphs that are constructed using Class Hierarchy Analysis (CHA) [13] or Rapid Type Analysis (RTA) [4] tend to grossly overapproximate the targets of virtual method calls. On the other hand, more precise callgraphs obtained using context-sensitive pointer analysis can take hours to construct. Even in cases where a sufficiently precise static callgraph can be efficiently constructed, it is useful to have some mechanism for specifying queries on the callgraph. Currently, client analyses that rely on callgraph information must implement their own ad-hoc analysis to answer application-specific queries about the callgraph.

In this paper, we address both of the above problems by presenting a new algorithm—and its implementation in a tool called EXPLORER—for automatically and precisely answering interprocedural control-flow queries about Java programs. EXPLORER allows programmers to formulate their queries about interprocedural control flow properties in the form of *regular expressions*. In particular, a query π for program P is true iff regular expression π can match a callstack prefix of P's possible executions. For instance, the expression `main` $\rightarrow .^* \rightarrow$ `foo` states that method `foo` is transitively reachable from `main`; such a query allows a user to determine whether `foo` is interprocedurally unreachable code.

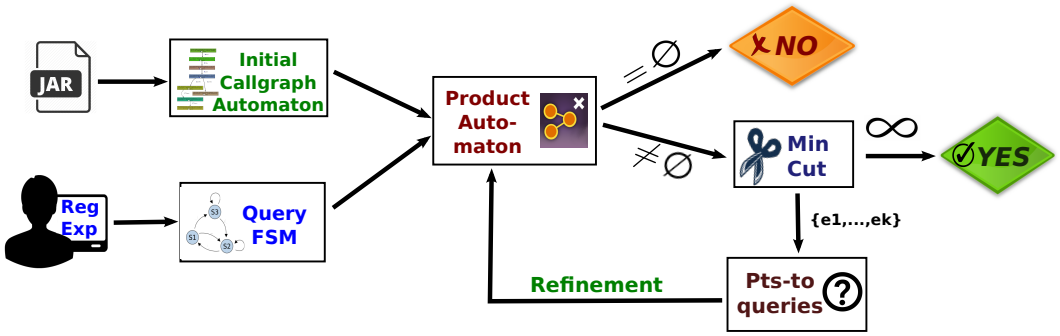

Figure 1. Overview of the EXPLORER algorithm for answering interprocedural control flow queries

Given a regular expression describing a set of possible callstack configurations, EXPLORER performs precise, query-driven static analysis and returns a yes/no answer indicating whether the specified callstack configuration is feasible. Towards this end, there are three important goals underlying the design of the EXPLORER tool:

1. *Soundness*: EXPLORER should return "no" *only if* the query describing a possible callstack configuration is infeasible for any possible program execution.
2. *Precision*: Since our algorithms are based on static analysis, EXPLORER may answer "yes" even though the property specified by the user is actually infeasible. However, our goal is to be as precise as possible, so EXPLORER should try to minimize the number false positives. To achieve this goal, EXPLORER constructs a precise callgraph by employing context-sensitive pointer analysis.
3. *Efficiency*: Since our techniques are meant to be used in an interactive setting—for instance, in the context of an IDE—EXPLORER should be sufficiently fast to be used in real time. Hence, rather than constructing a precise callgraph eagerly, EXPLORER refines only those callgraph edges that are relevant for answering a given query. Furthermore, EXPLORER is *refutation-based* and identifies the minimum number of callgraph edges that need to be refined to refute a query.

This paper presents the EXPLORER tool and describes its underlying ideas and algorithms. To demonstrate the applicability and practicality of our ideas, we evaluate EXPLORER by applying it to three different program analysis tasks, namely, (1) analysis of the observer design pattern in Java programs, (2) identification of performance bugs caused by GUI lagging, and (3) analysis of inter-component communication (ICC) in Android.

Contributions. In summary, this paper makes the following contributions:

- We present a simple but general query language for describing an important class of interprocedural control flow properties of object-oriented programs.
- We describe a practical algorithm for giving sound and precise answers to queries expressible in our framework. Our algorithm constructs an automaton that allows a given query to be checked against the application's callgraph. It then iteratively refines this automaton by using a demand-driven, context-sensitive pointer analysis, and it refutes only those callgraph edges that are relevant for answering the user's query.
- We evaluate the applicability and practicality of EXPLORER in the context of three different applications and show that it achieves a good tradeoff between precision and running time.

Organization. The rest of the paper is organized as follows. Section 2 gives a high-level overview of EXPLORER. We describe the syntax and semantics of our query language in Section 3 and present our algorithm in Sections 4, 5, and 6. The subsequent two sections then describe our implementation and evaluation. Finally, we discuss related work in Section 9 and conclude in Section 10.

2. Overview

Figure 1 shows an overview of the EXPLORER approach for answering interprocedural control flow queries about Java programs. EXPLORER takes two inputs: (1) the source or byte code of some Java application, and (2) a user-provided interprocedural control-flow query in the form of a regular expression. Given these two inputs, EXPLORER constructs a so-called *query automaton* as well as a *callgraph automaton*. The query automaton is simply an NFA representation of the user-specified regular expression, and the callgraph automaton is initially obtained from an imprecise callgraph of the application.

Next, EXPLORER constructs the product of the query and callgraph automata. However, since the resulting *product automaton* is typically huge, EXPLORER only constructs a *partial* product automaton $\mathcal{A}$ by pruning away many states that are guaranteed not to lead to an accepting state. If the language accepted by $\mathcal{A}$ is empty, we have *refuted* the query, meaning that the callstack configuration specified by the user is not feasible.

Query	Meaning
main $\rightarrow$ foo	Is `foo` called directly from `main`?
main $\rightarrow .^* \rightarrow$ foo	Is `foo` called directly or transitively from `main`?
$.^* \rightarrow$ foo $\rightarrow$ bar	Is `foo` a direct caller of `bar`?
$.^* \rightarrow$ foo $\rightarrow .^* \rightarrow$ bar	Can `foo` (directly or transitively) call `bar`?
$.^* \rightarrow$ foo $\rightarrow .^* \rightarrow$ (bar + baz)	Can `foo` (directly or transitively) call `bar` or `baz`?
$.^* \rightarrow$ (!foo)$\rightarrow$ bar	Is there a direct caller of `bar` other than `foo`?
$.^* \rightarrow$ (!(foo+baz)) $\rightarrow$ bar	Is there a direct caller of `bar` other than `foo` or `baz`?
((!foo)$^* \rightarrow$ bar)	Is it possible that `bar` is called in a context that does not involve `foo`?

Figure 2. Some example queries along with their meanings in English

On the other hand, if the language of $\mathcal{A}$ is non-empty, then there are two possibilities: Either the control-flow property queried by the user is indeed feasible, or the current callgraph is not precise enough to refute the query. Thus, EXPLORER proceeds to refine the product automaton by computing a *minimum cut* separating the initial states of the product automaton from the final states. If the cost of such a minimum cut is ∞, then no possible refinement of the callgraph can disprove the user's query, so EXPLORER has *validated*[1] the query and therefore produces an affirmative answer.

Conversely, if EXPLORER can find a finite-cost minimum cut separating the initial states from the accepting states, it then attempts to refute some of the existing transitions in the current product automaton. In particular, since the edges in the minimum cut correspond to potentially spurious targets of virtual method calls, EXPLORER extracts a set Q of relevant context-sensitive points-to queries. If it can refute each query $q \in Q$ using a demand-driven pointer analysis [33], then the language of the product automaton must be empty and the query has been refuted. On the other hand, even if all points-to queries in Q are validated, the answer to the user's query may still be "no", because there may be other cuts separating the initial states from the final states of the product automaton.

EXPLORER uses the answer to *each* points-to query to refine the product automaton. In particular, any query that is refuted by the pointer analysis is used to remove an existing transition from the latest product automaton, and any query q that is validated by the pointer analysis is used to update the *weight* of the transition associated with q. Since we set the weight of any validated transition to ∞, EXPLORER is guaranteed to never issue the same points-to query more than once. This strategy also ensures that the cost of the minimum cut becomes ∞ if no further refinement of the callgraph can allow the refutation of the user's query.

3. Query Language

We now explain the EXPLORER tool from a user's perspective and describe the syntax and semantics of its query language. Interprocedural control-flow properties in EXPLORER are specified using regular expressions in Grep-like syntax. For a given Java program P, EXPLORER accepts specifications written in the following query language:

$$\begin{array}{rcl} \textit{Query } Q & := & f \in \text{methods}(P) \\ & & \mid \ . \ \mid \ Q_1 \rightarrow Q_2 \ \mid \ !Q \\ & & \mid Q_1 + Q_2 \ \mid \ Q^* \ \mid \ Q^+ \ \mid \ (Q) \end{array}$$

The building blocks of queries are method names in program P, denoted $\text{methods}(P)$. The dot character (".") matches any method name, and the $\rightarrow$ operator indicates a call from one method to another. The exclamation mark operator ("!") is used to negate queries and the "+" operator is used for taking the disjunction of two queries. As usual, the "*" operator stands for Kleene closure, and Q^+ is syntactic sugar for $Q \rightarrow Q^*$.

More formally, to define the semantics of our query language, we first define a denotation function $[\![Q]\!]$ that maps queries to sets of strings:

$$\begin{array}{rcl} [\![f]\!] & = & \{f\} \\ [\![.]\!] & = & \text{methods}(P) \\ [\![(Q)]\!] & = & [\![Q]\!] \\ [\![!Q]\!] & = & \{f \mid f \notin [\![Q]\!] \wedge f \in \text{methods}(P)\} \\ [\![Q^+]\!] & = & \bigcup_{i=1}^{\infty} [\![Q_i]\!] \text{ where } \begin{array}{l} Q_1 = Q, \\ Q_i = Q \rightarrow Q_{i-1} \end{array} \\ [\![Q^*]\!] & = & \{\epsilon\} \cup [\![Q^+]\!] \\ [\![Q_1 + Q_2]\!] & = & \{s \mid s \in [\![Q_1]\!] \ \vee s \in [\![Q_2]\!]\} \\ [\![Q_1 \rightarrow Q_2]\!] & = & \{s_1 \cdot s_2 \mid s_1 \in [\![Q_1]\!] \wedge s_2 \in [\![Q_2]\!]\} \end{array}$$

In the last line, $s_1 \cdot s_2$ represents the concatenation of strings s_1 and s_2. Also, note that $!Q$ matches any function name f such that $f \notin [\![Q]\!]$.

Example 1. *To give the reader some intuition about our query language, Figure 2 shows some example queries along with their informal meanings in English. Observe that the query* $.^* \rightarrow$ foo $\rightarrow$ bar *is different from* foo $\rightarrow$ bar*: The latter query also states that foo is the entry method.*

In what follows, we define a *callstack* σ to be a string of the form $f_1 \cdot \ldots \cdot f_n$ where each f_i corresponds to the name of the i^{th} method currently pushed on the stack. We can now define the semantics of queries as follows:

[1] Since EXPLORER is based on static analysis, *validating* a query means failing to refute it using a certain abstraction of the program obtained through pointer analysis.

```
void main(...) {
  A x; A y;
  if(...) x = new A(); y = new B();
  else  x = new B(); y = new C();
  x.foo(); y.woo();
}

public A {
  void foo() {this.woo(); this.zoo(); this.bar();};
  void woo() {this.bar();};
  void bar() {System.out.println("In A:bar");};
  void zoo() {System.out.println("In A:zoo"};
}

public B extends A {
  void foo() {this.woo(); }
  void bar() {System.out.println("In B:bar");};
}

public C extends A {
  void bar() {System.out.println("In C:bar");};
}
```

Figure 3. Code example to illustrate our approach

Definition 1. (Query semantics) *Let Q be a query for program P. We say that Q evaluates to true if and only if there exists some $q \in [\![Q]\!]$ such that q is a prefix of a callstack σ that arises during some execution of P.*

Example 2. *Consider the code snippet shown in Figure 3, and the query* $(.^* \rightarrow \mathrm{A} : \mathrm{foo} \rightarrow .^* \rightarrow \mathrm{C} : \mathrm{bar})$*, where the notation $X : f$ denotes the f method in class X. The truth value of this query is* false *because the* bar *method of C can never be invoked from the* foo *method of* A *for the program shown in Figure 3.*

Since determining the truth value of an interprocedural control-flow query Q is undecidable, we require EXPLORER to over-approximate the answer to a query. That is, if EXPLORER returns false for a given query Q, then Q must indeed be false (i.e., soundness). However, the converse of this statement does not hold (i.e., incompleteness).

4. Callgraph and Query Automata

To answer a given query Q, EXPLORER first constructs the so-called *query automaton (QA)* and the *callgraph automaton (CGA)*. Here, the query automaton is simply an NFA-representation of the regular expression specified by the user. Since the problem of converting regular expressions to finite state machines is well-studied, we do not explain the QA construction in detail here.

Example 3. *Figure 4 shows the query automaton for the query from Example 2.*

To construct the callgraph automaton, EXPLORER first builds a sound but imprecise callgraph. Specifically, the

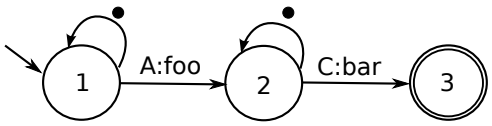

Figure 4. Automaton for query from Example 2

CGA for a given application P with respect to a callgraph $\mathcal{C}$ is a finite state machine $(S, \Sigma, \delta, q_0, F)$:

- The states S include all methods of P as well as a special state which we denote as χ. For a given method m, we denote the state representing m as q_m.
- The alphabet Σ consists of all methods of P.
- The transition function $\delta : S \times \Sigma \rightarrow S$ is obtained using callgraph $\mathcal{C}$. In particular, for each entry method m (without incoming edges) in $\mathcal{C}$, there exists a transition from state χ to q_m labeled with m. If $\mathcal{C}$ contains an edge from method m to method m', then the CGA contains a transition from q_m to q'_m labeled with m'.
- The initial state q_0 is the special state χ.
- The accepting states F include all states S.

Observe that the CGA accepts a given input word w if w corresponds to a valid callstack prefix according to the abstraction of the program given by callgraph $\mathcal{C}$.

Example 4. *Figure 5 shows the callgraph automaton for the code from Figure 3 with respect to a callgraph obtained using class hierarchy analysis. Due to lack of space, a state labeled q_{xy} denotes the method whose name starts with y in class X. For instance, q_{af} denotes the state associated with the* `foo` *method of class* `A`.

Since the initial CGA is constructed using an imprecise callgraph, it will need to be refined later on by removing certain spurious transitions. Since EXPLORER employs pointer analysis to refine the CGA, it is important to know which transitions are induced by which method calls in the program. For this purpose, we assume a function *Vars* which maps each transition $\tau \in \delta$ to a set of pairs of the form (v, T). Here, v is a variable name and T is a type. Specifically, $(v, T) \in \mathit{Vars}(\tau)$ if transition τ is induced by the assumption that variable v has dynamic type T.

Example 5. *Consider the CGA from Figure 5 and the code from Figure 3. Let τ be the transition from state q_m to q_{af}. Because of the method call* `x.foo()` *in* `main`*,* `main` *calls* `A:foo` *if* `x` *can have dynamic type* `A` *or* `C`*. Hence, we have* $\mathrm{Vars}(\tau) = \{(x, A), (x, C)\}$*. On the other hand, let τ' be the transition from state q_{af} to q_{az}. Here, we have* $\mathrm{Vars}(\tau') = \{(\mathrm{this}, A), (\mathrm{this}, B), (\mathrm{this}, C)\}$.

5. The Product Automaton

Having constructed the query and callgraph automata, the next step is to determine whether the intersection of the lan-

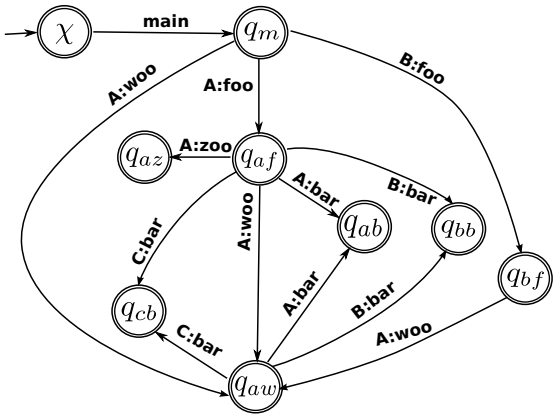

Figure 5. CGA for code from Figure 3

$$\frac{q_0 = \mathrm{Init}(CGA) \quad q_0' = \mathrm{Init}(QA)}{(q_0, q_0') \in \mathrm{States}(PA),\ \mathrm{Init}(PA) = (q_0, q_0')}$$

$$\frac{(q, q') \in \mathrm{States}(PA) \quad (q, x, q_1) \in \delta(CGA),\ \ (q', x, q_2) \in \delta(QA) \quad \mathcal{E}(q_1) \cap \mathcal{N}(q_2) = \emptyset}{(q_1, q_2) \in \mathrm{States}(PA),\ ((q, q'), x, (q_1, q_2)) \in \delta(PA)}$$

$$\frac{(q, q') \in \mathrm{States}(PA) \quad q \in \mathrm{Final}(CGA),\ \ q' \in \mathrm{Final}(QA)}{(q, q') \in \mathrm{Final}(PA)}$$

Figure 6. Partial PA construction algorithm. $\mathcal{N}(q)$ and $\mathcal{E}(q)$ indicate the necessary and error inputs for state q respectively. Also, $\delta(A)$ indicates the transition function for automaton A.

guages defined by the QA and the CGA is empty. Towards this purpose, we need to construct the *product automaton (PA)* for $\mathcal{L}(QA) \cap \mathcal{L}(CGA)$, where $\mathcal{L}(\mathcal{A})$ denotes the language accepted by automaton $\mathcal{A}$. However, since the callgraph automaton is typically very large for realistic Java software, explicitly constructing the entire product automaton may not feasible within a small time budget. Therefore, EXPLORER performs an optimized product construction that prunes irrelevant states of the PA on-the-fly.

The key idea underlying our algorithm is to exploit the fact that we are only interested in deciding the emptiness of the language defined by the product automaton. Specifically, recall that the language defined by the PA is empty if we cannot reach an accepting state from the initial state. Hence, if a certain state of the PA is not backwards-reachable from an accepting state, it can safely be pruned without changing the answer to a given query. Furthermore, since many methods in the callgraph are typically not relevant for answering a particular query, it is possible to quickly identify and prune away many irrelevant states of the product automaton.

To identify such irrelevant states in the product automaton, our algorithm starts by computing a set of *necessary inputs* for each state of the query automaton:

Definition 2. (Necessary input) *Given a finite state machine $(S, \Sigma, \delta, q_0, F)$, we say that a symbol $x \in \Sigma$ is a necessary input at some state $q^\star \in S$ if any word w that is accepted by the automaton $(S, \Sigma, \delta, q^\star, F)$ contains symbol x.*

In other words, a symbol x is a necessary input for state $q^\star$ if we *must* see symbol x in the remainder of the input before we can reach an accepting state. In the context of the query automaton, the necessary inputs for a state are all the methods that must be called to match the user's query.

Example 6. *Consider the query automaton from Figure 4. For the state labeled 1,* `A:foo` *and* `C:bar` *are both necessary inputs. On the other hand, for the state labeled 2, only* `C:bar` *is a necessary input.*

To construct the optimized product automaton, we also define so-called *error inputs*:

Definition 3. (Error input) *Given a finite state machine $(S, \Sigma, \delta, q_0, F)$, we say that a symbol $x \in \Sigma$ is an error input for some state $q^\star \in S$ if any word w containing symbol x is rejected by the automaton $(S, \Sigma, \delta, q^\star, F)$.*

Example 7. *Consider the CGA from Figure 5. For the states labeled q_{af}, q_{bf}, q_{aw},* `A:foo` *is an error input but* `C:bar` *is not. For the states labeled $q_{az}, q_{ab}, q_{bb}, q_{cb}$, both* `A:foo` *and* `C:bar` *are error inputs.*

To see why these concepts are useful, consider a state (q, q') in the product automaton such that $q \in$ *States*(CGA) and $q' \in$ *States*(QA). Now, let $\mathcal{N} = \{x_1, \ldots, x_n\}$ be the necessary inputs for state q' in the query automaton such that some $x_i \in \mathcal{N}$ is an error input for state q in the callgraph automaton. Since symbol x_i is necessary for acceptance in the query automaton but sufficient for rejection in the callgraph automaton, this means that state (q, q') can never lead to an accepting state in the product automaton. Hence, as soon as we encounter such a state (q, q') during the construction of the product automaton, we can prune it and avoid considering any states reachable from (q, q').

Figure 6 summarizes our optimized product automaton construction algorithm using inference rules. Here, $\mathcal{N}(q)$ denotes the necessary inputs for a state q and $\mathcal{E}(q)$ represents q's error inputs. For an automaton $\mathcal{A}$, $\mathrm{Init}(\mathcal{A})$ and $\mathrm{Final}(\mathcal{A})$ are $\mathcal{A}$'s initial and final states respectively, and $\mathrm{States}(\mathcal{A})$ represents all states of $\mathcal{A}$. According to the second rule of Figure 6, we do not add a state (q_1, q_2) to the product automaton if an error input for state q_1 in the CGA is a necessary input for a state q_2 in the QA.

Theorem 1. *Let $\mathcal{A}$ be the standard synchronous product of QA and CGA, and let $\mathcal{A}'$ the partial product automaton constructed according to Figure 6. Then, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

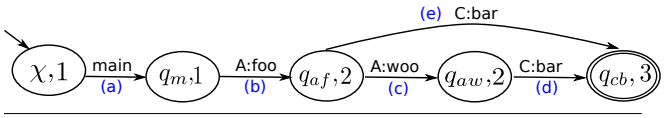

Figure 7. Partial PA constructed by EXPLORER

Proof. First, observe that every word that is accepted by $\mathcal{A}'$ is also accepted by $\mathcal{A}$ because every transition $\tau \in \delta(\mathcal{A}')$ is also in $\delta(\mathcal{A})$. Now, suppose there exists some word w that is accepted by $\mathcal{A}$ but rejected by $\mathcal{A}'$. Let $(q_0, q'_0), \ldots, (q_n, q'_n)$ be a run ϖ of $\mathcal{A}$ on input word w such that there is no corresponding run in $\mathcal{A}'$. This implies $\mathcal{E}(q_i) \cap \mathcal{N}(q'_i) \neq \emptyset$ for some $(q_i, q'_i) \in \varpi$. Now, let x be a symbol such that $x \in \mathcal{E}(q_i) \cap \mathcal{N}(q'_i)$. By definition of necessary input, the j'th symbol of w must be x for some $j > i$. Let $CGA = (S, \Sigma, \delta, q_0, F)$ and let w' be the suffix of w starting at position i. Since $x \in \mathcal{E}(q_i)$, by definition of error input, $q_i, \ldots, q_n$ cannot be a run of $(S, \Sigma, \delta, q_i, F)$ on w'. Hence, $(q_0, q'_0), \ldots, (q_i, q'_i), \ldots (q_n, q'_n)$ cannot be a run of $\mathcal{A}$ on input w (i.e., a contradiction). □

Example 8. *Figure 7 shows the partial product automaton constructed by* EXPLORER *for the QA and CGA from Figures 4 and 5. Note that many states that would appear in the full (standard) product automaton do not appear in the partial product automaton constructed by* EXPLORER *because they are pruned away due to the premise* $\mathcal{N}(q_1) \cap \mathcal{E}(q_2) = \emptyset$ *in the second rule of Figure 6.*

We now briefly describe how to compute necessary and error inputs. To compute error inputs for a state q in the CGA, we obtain all edge labels that are transitively reachable from q. If a given method m is not in this set, then m corresponds to an error input for state q. Similarly, we compute necessary inputs for a state q in the QA by collecting the set of symbols that appear in *all* acyclic paths from q to a final state.

6. Refinement Algorithm

We are now ready to describe our refinement-based query resolution algorithm, whose pseudo-code is shown in Algorithm 1. The query resolution algorithm takes as input the product automaton PA as well as the callgraph and query automata (CGA and QA) and returns a true/false answer to the user's query. As described in Section 2, our algorithm iteratively refines the product automaton until either the query is refuted or until no further refinement is possible.

The query resolution algorithm starts by associating a cost with each transition in the product automaton (lines 4-6). Intuitively, the cost associated with a transition estimates the difficulty of refuting a transition in the PA. In particular, a cost of ∞ indicates that a transition cannot be refuted, while a finite non-zero cost indicates that it *may* be possible to refute a certain caller-callee relation. Hence, Algorithm 1 calls a function called REFINABLE to check whether it *may be* possible to refute a transition. Specifically, REFINABLE(τ) returns false if and only if either the method call associated with τ is non-virtual (e.g., a static call) or if it already has a single target according to the initial imprecise callgraph. Hence, we assign a cost of ∞ to each transition that is not refinable and *unit cost* to all other transitions.[2]

Lines 7-24 of Algorithm 1 correspond to the main refinement loop. In each iteration, we first check the emptiness of the language defined by the product automaton (line 8). In graph-theoretic terms, this corresponds to checking the non-reachability of every final state from the initial state of the product automaton. If no final state is reachable, this means that the query can be refuted using our current abstraction of the callgraph; hence the algorithm returns false at line 9.

If the query cannot be refuted using the current product automaton, we try to refine it if possible. In particular, since our goal is to minimize the amount of work performed in each iteration, we would like to identify a minimum number of transitions that, if refuted, would be sufficient to refute the user's query. For this purpose, we compute a *minimum cut* of the product automaton. Specifically, at line 10 of the algorithm, we find a set of transitions $\mathcal{C}$ in the product automaton such that:

1. If $\mathcal{C}$ is removed from PA, every final state $f \in F$ would become unreachable from the initial state q_0.
2. The total cost of $\mathcal{C}$ is no greater than some other set of transitions $\mathcal{C}'$ satisfying condition 1.

Now, let $\mathcal{C}$ be such a minimum cut of the product automaton. If $\mathcal{C}$ is empty (i.e., the cost of the minimum cut is ∞), this indicates that no further refinement of the product automaton is possible. This happens, for example, when the pointer analysis fails to refute some of the points-to queries that are *necessary* for disproving a given control-flow query. In this case, the algorithm returns true at line 12, meaning that the answer to the user's query is "yes".

If there exists a finite-cost minimum cut $\mathcal{C}$, the algorithm then tries to refute each transition in $\mathcal{C}$ (lines 13-24). Note that, to prove the emptiness of $\mathcal{L}(\text{PA})$, we must refute all transitions in $\mathcal{C}$. For this purpose, for each transition $((q_1, q'_1), m, (q_2, q'_2)) \in \mathcal{C}$, we issue a set of points-to queries. In particular, to refute such a transition, we must prove the non-existence of a call from q_1 to q_2 under some calling context induced by q'_1, which corresponds to some prefix of the user's regular expression.

For this purpose, we first call the GETVARS function at line 15 to retrieve the set of pairs (v, T) such that q_1 calls q_2 if v has type T for *some* $(v, T) \in \text{Vars}(q_1, m, q_2)$ in the callgraph automaton.[3] Furthermore, since we are looking at

[2] While it may be possible to use more sophisticated heuristics for cost initialization, we have found this simple design choice to be sufficiently satisfactory in practice.

[3] Recall from Section 4 that *Vars* maps each transition in the CGA to a set of (v, T) pairs under which the corresponding call relationship is induced.

Algorithm 1 Refinement-Based Query Resolution

```
1:  procedure RESOLVEQUERY(PA, CGA, QA)
2:      Input: Product, callgraph, and query automata
3:      Output: true/false answer to query
4:      for all τ ∈ δ(PA)) do                          ▷ Init costs
5:          if REFINABLE(τ) then cost(τ) := 1
6:          else cost(τ) := ∞
7:      while true do
8:          if EMPTY(L(PA)) then
9:              return false;                          ▷ Query refuted
10:         Set C := MINCUT(PA, {q0}, F);
11:         if C = ∅ then
12:             return true;                           ▷ Query validated
13:         for all ((q1, q1'), m, (q2, q2')) ∈ C do
14:             Bool s := true;
15:             Set vt := GETVARS(CGA, (q1, m, q2));
16:             Context ctx := GETCTX(QA, q1');
17:             for all (v, T) ∈ vt do
18:                 Bool r := PTSQUERY(v, T, ctx)
19:                 if r then
20:                     cost((q1, q1'), m, (q2, q2')) := ∞;
21:                     s := false;
22:                     break;
23:             if s then
24:                 REMOVEEDGE((q1, q1'), m, (q2, q2'));
25:             else break;
```

a transition originating from state (q_1, q_1') in the product automaton, we want to know whether q_1 can call q_2 in a calling context satisfying state q_1'. Hence, we call the GETCTX function at line 16 to obtain the context associated with state q_1' in the query automaton. Specifically, if the query automaton is defined by $(S, \Sigma, \delta, q_0, F)$, then contexts satisfying q_1' are precisely those that are accepted by the automaton $(S, \Sigma, \delta, q_0, \{q_1'\})$.

Now, given variable v, type T, and relevant context ctx, our algorithm issues a points-to query to check whether variable v can have type T in context ctx (line 18). If this is indeed possible, this means that the transition under consideration is *not* spurious, and, hence, the query cannot be refuted using the current minimum cut $\mathcal{C}$. Thus, the algorithm updates the cost of the current transition to ∞ (line 20) and sets the boolean variable s indicating the spuriousness of the transition to false (line 21). Note that the effect of the two break statements at lines 22 and 24 is that the algorithm proceeds to find a new candidate refutation in the form of a new minimum cut.

On the other hand, if the pointer analysis can refute *every* points-to query associated with the current transition $\tau = ((q_1, q_1'), m, (q_2, q_2'))$, this means that τ is indeed spurious and we therefore remove it from the product automaton using the REMOVEEDGE procedure at line 23. If we are able to refute every transition $\tau \in \mathcal{C}$, the emptiness check at line 6 of the algorithm succeeds in the next iteration.

Example 9. *Consider the product automaton from Figure 7 and the code from Figure 3. Initially, the transitions labeled (a) and (c) in Figure 7 are assigned a cost of ∞, while all other edges have unit cost. Note that the edge labeled (c) is not "refinable" because there is already a single target of the call to* `woo` *according to the CHA-based callgraph.*

In the first iteration of the loop from Algorithm 1, the emptiness check at line 8 fails because the final state is reachable from the initial state. We therefore compute a minimum cut, which in this case yields a singleton containing edge (b). The variables associated with this transition are (`x`*, A), (*`x`*, C) and the context guard is empty. For the points-to query for (*`x`*, A), the answer is* true*, so we assign infinite cost to edge (b) and proceed to find a new refutation.*

In the next iteration, the emptiness check at line 8 fails again, so we find a new minimum cut, which now contains edges (e) and (d). The variable associated with edge (d) is (`this`*, C) in method* `woo`*. Furthermore, the context guard ctx includes* `A:foo`*; hence, we query whether* `this` *can have type C when* `woo` *is called from* `A:foo`*. This points-to query can be refuted using a context-sensitive pointer analysis, and we remove edge (d) from the automaton.*

In the next iteration of the inner loop (lines 13-24), we now try to refute edge (e). In this case, the variable of interest is `this` *in* `A:foo`*; hence we query whether* `this` *can have type* `C` *in* `foo`*. Since the result of this query is also negative, we also remove edge (e) from the automaton. In the next iteration, the emptiness check at line 8 succeeds, and the algorithm returns false as an answer to the query.*

Theorem 2. (Soundness). *If* EXPLORER *returns false for a query Q about program P, then Q must indeed evaluate to false on P.*

Proof. Suppose EXPLORER returns false, but there exists a callstack configuration $m_1, \ldots, m_n$ that arises in some execution of P and that matches Q. By soundness of the initial callgraph, $m_1, \ldots, m_n$ must be accepted by the CGA. Since $CGA \times QA$ accepts $\mathcal{L}(CGA) \cap \mathcal{L}(QA)$, Theorem 1 implies that the initial product automaton $\mathcal{A}_0$ passed as input to RESOLVEQUERY must accept the word $m_1, \ldots, m_n$. Hence, RESOLVEQUERY must have removed a transition $\tau = ((q_{m_{i-1}}, q_{i-1}'), m_i, (q_{m_i}, q_i'))$ from the PA such that $\mathcal{A}_0$ takes transition τ on the i'th input symbol of $m_1, \ldots, m_n$. Since τ was removed by RESOLVEQUERY, the pointer analysis must state that m_{i-1} cannot call m_i in a context satisfying state q_{i-1}' of $QA = (S, \Sigma, \delta, q_0', F)$. Since the pointer analysis is sound, this means that $(S, \Sigma, \delta, q_0', \{q_{i-1}'\})$ does not accept $m_1, \ldots, m_{i-1}$. However, this implies that τ cannot be the i'th transition taken by $\mathcal{A}_0$ on the input word $m_1, \ldots, m_n$ (i.e., a contradiction). □

7. Implementation

We implement our ideas in a new tool called EXPLORER[4] which consists of approximately 8,000 lines of Java code. EXPLORER is built on top of the Soot framework [23, 37] and accepts Java source code as well as Java and Dalvik byte code as input. To build the initial callgraph automaton for an application A, we first construct a harness that has a single entry method called m and which explicitly invokes every original entry method in A. We then run a scalable, but context-insensitive pointer analysis provided by Soot [23] to construct a callgraph, which is then used to generate the initial CGA. We convert user-provided regular expressions to query automata using JSA [9], which is a general tool for performing string analysis of Java programs. Since we introduce a new method called m during harness construction, EXPLORER also instruments each user-provided query Q and rewrites it as $m \rightarrow Q$.

To implement the refinement procedure of Algorithm 1, we use the Ford-Fulkerson algorithm [17] for finding minimum cuts. We also use the context-sensitive demand-driven pointer analysis of Sridharan and Bodík for answering points-to queries issued by RESOLVEQUERY [33]. Although Sridharan and Bodík's original tool is capable of answering points-to queries under different contexts, they do not expose the corresponding interface in their Soot implementation (i.e., their implementation always passes an empty context). Since our RESOLVEQUERY procedure needs to answer points-to queries under a context guard, we added a new interface to Sridharan and Bodík's implementation that also takes the calling context as a parameter. In our queries, we limit the length of context to be one. Finally, Sridharan and Bodík's tool requires the user to specify a time budget for each points-to query, and we use the default setting, which refines at most 75,000 nodes of the pointer assignment graph per query. Queries that need to traverse more than 75,000 nodes terminate and return a conservative result.

In addition to giving yes/no answers to queries, EXPLORER can also provide so-called *witnesses*. A *witness* is a callstack prefix that (1) matches the user-provided regular expression, and (2) is feasible according to the refined callgraph. While we do not utilize these witnesses in our experimental evaluation, we believe they are useful in various software engineering and program analysis tasks.

8. Evaluation

To evaluate the usefulness and practicality of EXPLORER, we apply it to three different analysis tasks:

1. Analysis of the observer design pattern in Java programs
2. Identification of performance bugs caused by GUI lagging in Android applications
3. Analysis of inter-component communication in Android

[4] Please see http://fredfeng.github.io/explorer/

Benchmark	# queries	CHA	CI	KOBJ	EXPLORER
batik	101	1	1	11	13
fop	261	97	111	137	165
sunflow	76	50	52	54	54
weka	165	21	21	52	90
jmeter	300	116	140	N/A	210

Table 1. Number of refutations for resolving queries related to the observer design pattern (higher is better).

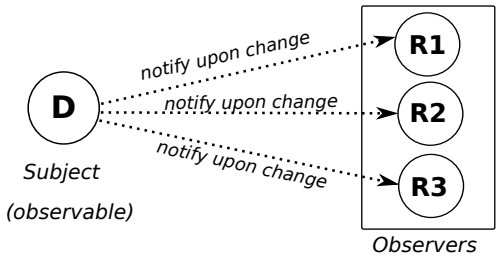

Figure 8. Schematic illustration of the observer pattern

In addition to showing that EXPLORER can be useful in these different scenarios, we also answer the following research questions:

- How useful is the refinement-based algorithm compared to its eager versions using different callgraph construction schemes?
- What is the benefit of the optimized product automaton construction described in Section 5 compared to the standard product automaton construction?
- Is it useful to employ minimum cuts to refine the PA?

In all experiments, we analyzed JDK1.6.0_45 as well as other third-party libraries that are used by our benchmarks. All experiments were conducted on an Intel Xeon(R) computer with an E5-2630 CPU and 64G of memory running on the Ubuntu 14.04 operating system.

8.1 Analysis of the Observer Design Pattern

In our first experiment, we use EXPLORER to analyze the *observer design pattern* in Java programs [12]. The observer paradigm is a common Java design pattern that allows a given object o to notify its dependents about modifications to o (see Figure 8). Because of the ubiquity of this design pattern, the Java SDK provides an `Observer` interface that allows different classes to specify what action to perform upon changes to the observed object.

A common question that arises in many automated testing, program understanding, and analysis tasks is whether instances of a given class A can be observers of objects of type B [8, 36, 39]. Precise answers to this question are particularly important for understanding event-driven applications and GUI code. Fortunately, such queries are easy to formulate in the EXPLORER framework. For instance,

Benchmark	CHA	CI	KOBJ	EXPLORER	EXPLORER(NoCut)	EXPLORER(Naive PA)
batik	28+94	64+19	6812+15	64+72	T/O	64+6255
fop	26+260	59+33	6710+31	59+139	T/O	T/O
sunflow	20+73	51+10	3821+12	51+54	T/O	51+1011
weka	31+200	45+26	3891+22	45+325	T/O	45+3392
jmeter	32+365	78+81	T/O	78+1188	T/O	T/O

Table 2. Running time for answering queries related to the observer design pattern. Times are indicated in the format $x + y$, where x is the time in seconds to construct the initial call graph and y denotes the time (s) for answering queries. "T/O" denotes time-out after 2 hours.

consider an application that uses the `Observer` interface provided by the Java SDK such that class `A` inherits from the `Observable` base class, and class `B` implements the `Observer` interface. Now, to determine whether class `B` is an observer for class `A`, we can issue the following query:

$$.^* \rightarrow \texttt{A} : \texttt{notifyObservers} \rightarrow .^* \rightarrow \texttt{B} : \texttt{update}$$

In other words, we check whether the `notifyObservers` method of class `A` can transitively call `B`'s `update` method.

Thus, to perform our first experiment, we manually identify the observable and observer interfaces provided by the libraries used in our experimental benchmarks. For example, in the Java Swing library, an observer interface is `ActionListener` which requires an `actionPerformed` method. Hence, to determine whether `XListener` is an observer of a specific Swing component called `YButton`, we issue a query of the form:

$$.^* \rightarrow \texttt{YButton} : \texttt{fireActionPerformed} \rightarrow .^* \rightarrow \texttt{XListener} : \texttt{actionPerformed}$$

In what follows, we discuss the precision and performance of EXPLORER when answering queries of this form on the five benchmarks shown in the first column of Table 1. Three of these programs (namely *batik*, *fop*, *sunflow*) are from DaCapo [7]; *weka* is a well-known tool for data mining and machine learning, and *jmeter* is a widely-used tool for functional and performance testing of Java programs.[5]

Refinement based vs. eager algorithm. Recall that one of the design choices underlying EXPLORER is to refine the callgraph on demand rather than to construct it eagerly. To investigate this design choice, we compare EXPLORER against a version of itself (let's call it EXPLORER') that does not use a refinement-based approach. Instead, EXPLORER' constructs the product automaton $\mathcal{A}$ using the initial callgraph, and it returns false if and only if the language of $\mathcal{A}$ is empty. We write EXPLORER'(C) to denote the eager version of EXPLORER using callgraph construction scheme C. For example, EXPLORER'(CI) and EXPLORER'(KOBJ) denote the eager versions of EXPLORER where the callgraph is constructed using context-insensitive and k-object-sensitive pointer analysis [26] respectively.

Table 2 shows the running times of EXPLORER and EXPLORER'(C) for different callgraph construction schemes when answering queries of the form "Is class `A` an observer of class `B`?". All running times are in seconds and are written using the format $x + y$, where x denotes the time to build the initial callgraph, and y is the time to answer queries. In the case of EXPLORER, the time to answer queries (i.e., y) includes refinement, points-to queries, construction of minimum cuts, construction of the product automaton, and the tests for emptiness. In the case of EXPLORER'(C), the time to answer queries only includes the time to construct the product automaton and test its emptiness. In Table 1, we compare the precision of EXPLORER with EXPLORER'(C) in terms of the number of queries refuted. Thus, higher numbers in Table 1 indicate better precision.

First, we compare EXPLORER against EXPLORER'(CHA) and EXPLORER'(CI), where CI is the context-insensitive subset-based pointer analysis implemented in Soot [23]. Since EXPLORER uses the same context-insensitive pointer analysis to construct the *initial* callgraph, EXPLORER'(CI) corresponds exactly to EXPLORER without any refinement. As expected, we see that EXPLORER'(CI) is faster but also significantly less precise than EXPLORER: On average, EXPLORER can refute 63.7% more queries than EXPLORER'(CI). Observe that EXPLORER'(CHA) is also generally faster compared to EXPLORER but even less precise.

Let us now compare EXPLORER with EXPLORER'(KOBJ) for $k = 1$ (i.e., 1-object-sensitive pointer analysis). From Tables 1 and 2, we see that EXPLORER is superior to EXPLORER'(KOBJ) in terms of both precision and running time. In particular, since eager callgraph construction using context-sensitive pointer analysis is much more expensive than EXPLORER's initial context-insensitive analysis and subsequent refinement, EXPLORER has significant advantage over EXPLORER'(KOBJ) in terms of running time when answering all observer-related queries. We do not compare EXPLORER with EXPLORER'(KCFA) because EXPLORER'(KCFA) is both less precise and slower than EXPLORER'(KOBJ) on these benchmarks.[6]

Impact of optimized product construction and min cuts.

[5] We do not use all DaCapo benchmarks since many of them do not heavily use the observer pattern.

[6] EXPLORER'(KCFA) denotes an eager version of EXPLORER where the initial callgraph is obtained using a k-CFA pointer analysis [31].

In Table 2, the column labeled "EXPLORER(NoCut)" shows the performance of EXPLORER when it refines all edges in the product automaton instead of computing a minimum cut $\mathcal{C}$ and refining only those edges in $\mathcal{C}$. (This version still uses the optimized product construction.) We see that without the use of minimum cuts, the tool does not compute an answer for any benchmark within our two hour time limit. The last column of Figure 2, which is labeled "EXPLORER(NaivePA)," shows the running time of EXPLORER when it performs the standard product automaton construction instead of the optimized construction described in Section 5. Here, we see that our optimized product automaton construction has a significant positive impact on EXPLORER's performance.

8.2 Detection of Performance Bugs

In our second experiment, we use EXPLORER to automatically detect performance bugs caused by GUI lagging in Android applications. As described in the PerfChecker paper [24], many of the performance bugs in Android applications occur when some lengthy operation is performed directly in the main thread. To detect performance bugs in Android, the developers of the PerfChecker tool have identified a set of API methods, such as `openConnection`, `query`, and `openFileInput`, that perform potentially lengthy operations. Now, let L be a regular expression that describes the disjunction of these operations (i.e., `openConnection + query + ...`). Similarly, let M be a regular expression that describes possible *lifecycle methods*[7] (e.g., `onStart`, `onCreate`, etc.) of Android activities.

A GUI lagging performance bug arises if any method in M can call any method $l \in L$ without starting an asynchronous task that performs l in the background. Assuming A is a regular expression describing methods that start asynchronous tasks (e.g, `AsyncTask::doInBackground`, `Thread::run` etc.), the following regular expression characterizes the condition under which a GUI lagging performance bug will occur:

$$.^* \rightarrow M \rightarrow (!A)^* \rightarrow L$$

In other words, some $m \in M$ transitively calls some $l \in L$ without spawning a new thread or asynchronous task. Given this regular expression, we can now use EXPLORER to automatically detect GUI lagging performance bugs in Android applications!

Precision. We now consider the precision of EXPLORER on all 18 benchmarks from the PerfChecker paper [24]. These results are summarized in Table 3, where a ✓ indicates that the tool reported a performance bug, and a ✗ indicates that it did not. According to the PerfChecker paper, the first twelve applications in Table 3 contain GUI lagging problems, while the last six do not. EXPLORER identifies the bugs in all twelve buggy applications and does not report false alarms for any of the remaining six. We manually confirmed that the warnings generated by EXPLORER but not Perfchecker represent true positives. In particular, the performance bugs in Firefox and Bitcoin Wallet arise in library code, which PerfChecker does not analyze.

Benchmark	CHA	CI	KOBJ	EXP	PerfC
Ushahidi	✓	✓	✓	✓	✓
c:geo	✓	✓	✓	✓	✓
Omnidroid	✓	✓	✓	✓	✓
Geohash Droid	✓	✓	✓	✓	✓
Android Wifi Tether	✓	✓	✓	✓	✓
Osmand	✓	✓	T/O	✓	✓
WebSMS	✓	✓	✓	✓	✓
ConnectBot	✓	✓	✓	✓	✓
Firefox	✓	✓	✓	✓	✗
APG	✓	✓	✓	✓	✓
FBReaderJ	✓	✓	✓	✓	✓
Bitcoin Wallet	✓	✓	✓	✓	✗
Open GPS Tracker	✗	✗	✗	✗	✗
My Tracks	✓	✓	T/O	✗	✗
XBMC Remote	✓	✗	✗	✗	✗
AnySoftKeyboard	✗	✗	✗	✗	✗
OI File Manager	✓	✓	✗	✗	✗
IMSDroid	✗	✗	✗	✗	✗

Table 3. GUI lagging bug detection results. ✓denotes that the app has at least one performance bug. EXP denotes EXPLORER, and PerfC stands for PerfChecker.

In comparing the precision of the eager versions EXPLORER'(C) for different choices of callgraph construction algorithms C, we see that for the six non-buggy benchmarks, both EXPLORER'(CHA) and EXPLORER'(CI) have $> 30\%$ false positive rate. On the other hand, EXPLORER'(KOBJ) does not report any false positives but times out for two of the 18 benchmarks (and performs quite poorly in terms of running time for some of the other benchmarks; see Table 5).

Table 4 presents a more detailed view of the same experiment. In particular, as a proxy for the number of performance bugs reported by each version of the tool, we now issue $|M| \times |L|$ different queries of the form $.^* \rightarrow m \rightarrow (!A)^* \rightarrow l$ where $m \in M$ and $l \in L$. In other words, each such query checks whether a given life cycle method can call some particular lengthy operation. According to the data in Table 4, EXPLORER reports fewer bugs than each eager version EXPLORER'(C). Furthermore, in a few cases (e.g., Ushahidi), EXPLORER reports many fewer bugs than EXPLORER'(KOBJ), which suggests that the context-sensitive points-to queries issued by EXPLORER's refinement algorithm are critical in some cases. (Recall from Section 6 that

[7] A *lifecycle* method of a component is invoked by the Android SDK when the application transitions between different states of its lifecycle. For example, they are called for starting, pausing, or resuming an activity.

Benchmark	CHA	CI	KOBJ	EXPLORER	EXPLORER(NoCut)	EXPLORER(Naive PA)
Ushahidi	5 + 23	13+10	234+10	13+10	13+16	13+29
c:geo	4 + 47	39+18	1824+14	39+18	39+29	39+301
Omnidroid	1+9	3+5	61+5	3+7	3+10	3+21
Geohash Droid	1+3	3+2	11+2	3+2	3+2	3+7
Android Wifi Tether	1+1	4+1	5+1	4+1	4+1	4+5
Osmand	8+163	41+14	T/O	41+15	41+97	41+189
WebSMS	2+13	10+3	171+3	10+4	10+5	10+12
ConnectBot	1+4	4+2	14+2	4+2	4+2	4+12
Firefox	11+68	55+17	3705+15	55+18	55+19	55+38
APG	7+108	41+12	119+12	41+12	41+18	41+29
FBReaderJ	5+23	26+10	372+10	26+11	26+123	26+36
Bitcoin Wallet	10+71	52+14	3607+13	52+15	52+18	52+28
Open GPS Tracker	5 + 19	17+8	307+9	17+9	17+12	17+10
My Tracks	12+362	61+16	T/O	61+17	61+20	61+29
XBMC Remote	3+11	12+5	173+4	12+6	12+6	12+17
AnySoftKeyboard	1+5	2+3	10+3	2+3	2+4	2+14
OI File Manager	1+3	2+3	8+2	2+3	2+3	2+16
IMSDroid	3+18	12+9	91+10	12+10	12+14	12+28

Table 5. Running time for detecting performance bugs. Times are indicated in seconds and using the format $x + y$, where x is the time to construct the initial call graph and y is the time for answering queries. "T/O" denotes time-out with a time limit of 2 hours.

Benchmark	CHA	CI	KOBJ	EXPLORER
Ushahidi	91	15	15	9
c:geo	179	51	51	42
Omnidroid	27	27	27	27
Geohash Droid	13	3	3	3
Android Wifi Tether	4	4	1	1
Osmand	258	51	N/A	51
WebSMS	1	1	1	1
ConnectBot	14	10	10	9
Firefox	85	4	4	4
APG	60	50	45	39
FBReaderJ	41	11	11	11
Bitcoin Wallet	71	1	1	1
Open GPS Tracker	0	0	0	0
My Tracks	321	3	N/A	0
XBMC Remote	17	0	0	0
AnySoftKeyboard	0	0	0	0
OI File Manager	2	2	0	0
IMSDroid	0	0	0	0

Table 4. Number of performance bugs (lower is better).

EXPLORER's refinement algorithm extracts the relevant context associated with a given automaton state.)

Refinement-based vs. eager. To evaluate the benefits of the refinement-based approach, Table 5 compares the running times of EXPLORER with those of EXPLORER'(C) for different callgraph construction algorithms. We see that EXPLORER'(CHA) is often faster than EXPLORER, even though there are some cases (e.g., My Tracks, Osmand, APG) where EXPLORER performs better than EXPLORER'(CHA). As in Section 8.1, we see that EXPLORER'(CI) is faster than EXPLORER, but at the cost of decreased precision (see Tables 3 and 4). On the other hand, EXPLORER'(KOBJ) often has poor performance because of the cost of performing eager whole-program context-sensitive pointer analysis.

Impact of min cuts and optimized PA construction. The column labeled "EXPLORER(No Cut)" in Table 5 shows the running time of EXPLORER when it refines all edges in the product automaton instead of computing a minimum cut. Here, the impact of using minimum cuts is not as dramatic as it was in Section 8.1. In fact, for two benchmarks (Ushahidi and IMSDroid), the version of EXPLORER that does not use minimum cuts performs slightly better because these benchmarks require all edges to be refined, so the minimum cut computation only adds overhead. For other benchmarks, such as Osmand, MyTracks, and FBReaderJ, the use of minimum cuts has a substantial positive impact.

The column labeled "EXPLORER(Naive PA)" in Table 5 shows the running time of EXPLORER when it uses the standard product construction algorithm (i.e., without the technique of Section 5). For many applications, we see that our optimized product automaton construction significantly improves performance.

8.3 Android Inter-Component Communication

In our third experiment, we use EXPLORER to reason about inter-component communication (ICC) in Android applications. In particular, Android applications are composed of different kinds of components that can asynchronously invoke each other. For instance, a GUI component A might start a service component B when the user clicks on a certain

```
1. Activity A {
2.    onCreate(...) { foo(); }
3.    foo(){
4.       Intent n = new Intent();
5.       n.setClass(B.class); //ICC site
6.       startActivity(n);
7.    }
8. }
```

Figure 9. An ICC example.

button on the screen. ICC analysis determines which components in an Android application can start which other components. Such an analysis has many applications in test case generation, bug finding, and malware detection [3, 16, 28]: For example, the malware detection tool described in [16] uses inter-component calls as part of its malware signature and performs static ICC analysis to determine whether an Android application belongs to a malware family. As another example, the A^3E tool described in [3] performs static ICC analysis to construct a so-called *activity transition graph*, which is used to increase coverage and efficiency for automated testing of Android applications.

Figure 9 gives a simple example demonstrating Android ICC. Here, a component called `A` starts another component called `B` by invoking the Android SDK's `startActivity` method. In general, precise reasoning about ICC requires two different analyses:

- ***Control-flow analysis:*** Determine if an *ICC site* (e.g., call to `startActivity`) is transitively reachable from a component's *life-cycle methods* (e.g., `onCreate`).
- ***Data-flow analysis:*** Identify the values of the fields of the `Intent` object that are provided as an argument to an ICC method (e.g., `startActivity`).

Going back to our example from Figure 9, component `A` can start component `B` because (1) the `onCreate` method of `A` can transitively call `startActivity`, thereby satisfying the control-flow requirement, and (2) the field of the intent object associated with this ICC site refers to `B` (hence satisfying the data-flow requirement).

The control-flow analysis required for ICC resolution can be naturally formulated in the EXPLORER framework.[8] Specifically, let $m_1, \ldots, m_n$ be the life-cycle methods (e.g., `onCreate`) of a component and let m' be a method that contains an ICC site. Then, the control flow aspect of ICC resolution answers the following query:

$$.^* \rightarrow (m_1 + \ldots + m_n) \rightarrow .^* \rightarrow m'$$

To evaluate the precision and performance of EXPLORER in this context, we generate many ICC-related control-flow queries for several popular Android applications that are available through Google Play. Most of these applications

[8] We do not address the data-flow component of ICC, as it mainly requires string analysis, which is orthogonal to this paper.

Benchmark	CHA	CI	KOBJ	EXPLORER
com.twitter	244	600	N/A	678
com.yelp	0	354	N/A	354
com.snapchat	8	140	N/A	168
com.whatsapp	39	290	N/A	290
com.netflix	112	275	N/A	380
com.pandora	0	180	N/A	180
com.instagram	121	541	N/A	567
com.expedia	63	471	N/A	553
com.pinterest	83	322	N/A	346
com.yahoo.mail	196	400	N/A	439
com.abcnews	171	318	N/A	350
com.walmart	163	273	N/A	301

Table 6. Number of refutations among 1000 ICC-related queries. Higher numbers indicate better precision.

are notoriously complicated and pose scalability problems for static analysis techniques.

Precision and performance. Tables 6 and 7 compare the precision of and running time of EXPLORER against those of EXPLORER'(C) for different callgraph construction algorithms. First, comparing EXPLORER and EXPLORER'(CHA), we see that EXPLORER is not only much more precise but sometimes even faster compared to EXPLORER'(CHA) (e.g., netflix, pinterest). As expected, EXPLORER'(CI) is always faster than EXPLORER, but it is less precise for most of the benchmarks. In particular, the context-insensitive analysis refutes 347 (out of 1000) queries on average, while EXPLORER refutes 384 queries. Finally, observe that the running time of EXPLORER is quite moderate; the average time to answer one ICC-related query is 0.24s.

Impact of min cut and optimized PA construction. As we see from the last two columns of Table 7, the use of minimum cuts and the optimized product automaton construction are both crucial in this application domain. In particular, if we refine all edges of the product automaton instead of only those in the minimum cut, the tool fails to terminate within the two hour time-limit. Similarly, if we use the standard product automaton construction, the modified EXPLORER fails to terminate within two hours. These statistics indicate that the optimized PA construction and the use of minimum cuts are both critical to the scalability and practicality of EXPLORER.

9. Related Work

EXPLORER provides the first general framework for answering a broad class of interprocedural control flow queries about feasible callstack configurations. However, EXPLORER bears similarities to other program analysis techniques that are query- or demand-driven.

Analysis generators from high-level specifications. At a high-level, EXPLORER can be viewed as a tool for auto-

Benchmark	CHA	CI	KOBJ	EXPLORER	EXPLORER(NoCut)	EXPLORER(Naive PA)
com.twitter	7 + 119	15 + 32	T/O	15 + 159	T/O	T/O
com.yelp	12+155	30+47	T/O	30+343	T/O	T/O
com.snapchat	2 + 55	21 + 4	T/O	21 + 11	T/O	T/O
com.whatsapp	11+198	55+45	T/O	55+767	T/O	T/O
com.netflix	6 + 111	18 + 14	T/O	18 + 55	T/O	T/O
com.pandora	8 + 174	51 + 43	T/O	51 + 365	T/O	T/O
com.instagram	10 + 294	45 + 51	T/O	45 + 96	T/O	T/O
com.expedia	7+154	12+35	T/O	12+116	T/O	T/O
com.pinterest	10+443	40 + 15	T/O	40 + 54	T/O	T/O
com.yahoo.mail	4+138	25+55	T/O	25+333	T/O	T/O
com.abcnews	8+206	43+31	T/O	43+142	T/O	T/O
com.walmart	8+147	62+28	T/O	62+65	T/O	T/O

Table 7. Running time for answering 1000 ICC-related queries. Running times are indicated in the format $x + y$, where x is the time in seconds to construct the initial call graph and y denotes the time for answering queries. T/O indicates time-out with a time limit of 2 hours.

matically generating analyses from user-specified queries. In this regard, EXPLORER is similar to other systems, such as SLIC [6], Metal [15], PQL [25], Rhodium [22] etc. Among these systems, SLIC and Metal provide declarative event-based specification languages for building system-specific static checkers [6, 15]. The PQL system [25] allows users to specify code patterns and generates a dynamic analysis that detects matches at run-time and performs remedial actions. PQL also utilizes static analysis to reduce run-time overhead. The Rhodium system [22] provides a language for specifying dataflow analyses that can then be automatically proven sound. In contrast to all of these systems, EXPLORER allows users to specify control-flow queries about possible callstack configurations and employs demand-driven techniques for answering these queries.

Automata-based techniques and model checkers. Regular expressions and automata have a long history of use as specifications in program analysis and verification [6, 15, 29, 32]. However, unlike previous techniques which use automata to specify typestate properties [35] (or variations thereof), we use regular expressions to describe feasible callstack configurations of programs.

Finite-state and Büchi automata also have numerous applications in model checking finite and infinite-state systems [5, 11]. Specifically, many automata-theoretic model checking techniques describe both the design and specification of a system as Büchi automata and compose them using the product construction. EXPLORER bears similarities to model checking techniques in that it composes the query automaton with the callgraph automaton and checks the emptiness of the resulting product automaton.

Counterexample Guided Abstraction Refinement. EXPLORER's method of iteratively refining the callgraph bears some similarity to the notion of counterexample guided abstraction refinement (CEGAR) used in model checking [10, 21]. Both CEGAR and EXPLORER start with imprecise abstractions of the program—a predicate abstraction for CEGAR and a callgraph for EXPLORER—and both iteratively and selectively refine these abstractions to prune infeasible paths from the abstraction. Conceptually, the two approaches differ in that CEGAR uses under-approximations to refine the abstraction, while EXPLORER uses (more precise) over-approximations to refine the callgraph.

Demand-driven program analysis. Demand-driven program analysis techniques differ from *eager (exhaustive) analyses* in that they only analyze parts of the program that are relevant for answering a given query. The idea of demand-driven program analysis originates from the work of Duesterwald et al. for interprocedural dataflow analysis [14] and has found numerous applications in pointer analysis [19, 20, 33, 34, 38, 40]. The algorithm employed in EXPLORER is also demand-driven and utilizes the context-sensitive pointer analysis proposed by Sridharan and Bodík [33].

Callgraph construction. EXPLORER answers interprocedural control-flow queries with respect to a callgraph abstraction of the program. There are many different techniques for constructing the callgraph of Java programs. One of the simplest techniques is class hierarchy analysis (CHA), which examines the inheritance relationship between classes to overapproximate targets of virtual method calls [13]. Rapid type analysis (RTA) refines CHA by pruning methods whose enclosing classes have not been instantiated as possible receiver types [4]. More precise callgraph construction algorithms utilize pointer analysis by querying possible dynamic types of the receiver. While EXPLORER obtains an initial callgraph using context-insensitive pointer analysis, it performs refinement using context-sensitive techniques.

Agrawal et al. [1] present a demand-driven callgraph construction algorithm that identifies the possible targets of a polymorphic callsite. However, unlike EXPLORER, their technique does not allow users to specify queries about interprocedural control flow properties.

Ali et al. describe a precise callgraph construction algorithm [2], but because their approach is eager, it does not scale as well as EXPLORER. Thus, for example, they do not attempt to analyze library code.

Control flow analysis and applications. Interprocedural control flow analysis determines the set of functions that can be referred to by a given expression and is useful for both functional and object-oriented programs [27, 30]. EXPLORER allows answering a class of interprocedural control flow queries of object-oriented programs that are expressible as regular expressions. As demonstrated in Section 8, a general framework for answering interprocedural control-flow queries has many applications in program understanding and analysis. In addition to being useful in various program analysis tasks such as analysis of Android ICC [28], performance bug identification [24], and malware detection [16], we also envision EXPLORER being very useful to end users in the context of IDEs such as Eclipse and Visual Studio.

10. Conclusion

We have described a general framework for statically answering a class of interprocedural control flow queries specified as regular expressions. We have implemented this idea in a tool called EXPLORER and demonstrated that it is useful for a variety of program analysis and understanding tasks. Our experimental evaluation demonstrates that EXPLORER is both precise and practical. Our results also highlight the advantages of performing refinement-based analysis as well as the importance of our partial product automaton construction and the use of minimum cuts.

In future work, we plan to explore the applicability of EXPLORER for other clients and implement an Eclipse plug-in that allows programmers to use EXPLORER for software understanding tasks while programming. We also plan to extend the ideas proposed in this paper and design an even more general framework for answering a combination of interprocedural control-flow and dataflow queries.

Acknowledgments

We thank Saswat Anand for his suggestion regarding our third experiment and Manu Sridharan for helping us add a new interface to his demand-driven pointer analysis in Soot. We also thank Thomas Dillig, Navid Yaghmazadeh, Jia Chen, Osbert Bastani and Stefan Heule for their insightful feedback. Finally, we would like to thank the anonymous reviewers for their thorough and helpful comments.

This research is sponsored in part by the Air Force Research Laboratory under agreement numbers FA8750-12-2-0020 and FA8750-14-2-0270 and in part by NSF Awards #1453386, CNS-1138506 and DRL-1441009. The views, opinions, and findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] G. Agrawal, J. Li, and Q. Su. Evaluating a demand driven technique for call graph construction. In *CC*, pages 29–45. Springer, 2002.

[2] K. Ali and O. Lhotak. Application-only call graph construction. In *ECOOP*, pages 688–712. ACM, 2012.

[3] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, pages 641–660, 2013.

[4] D. Bacon and P. Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31:324–341, 1996.

[5] C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[6] T. Ball and S. K. Rajamani. Slic: a specification language for interface checking (of c). Microsoft Research, 2002.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[8] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18, 2003.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Journal of the ACM*, 50(5):752–794, 2000.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[12] J. W. Cooper. *Java design patterns: a tutorial*. Addison-Wesley Professional, 2000.

[13] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.

[14] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *POPL'95*, pages 37–48.

[15] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*. USENIX Association, 2000.

[16] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *SIGSOFT FSE*, 2014.

[17] L. Ford and D. R. Fulkerson. *Flows in networks*, volume 1962. Princeton Princeton University Press, 1962.

[18] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys*, pages 281–294, 2012.

[19] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Static Analysis*, pages 214–236. Springer, 2003.

[20] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI*, pages 24–34. ACM, 2001.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN Workshop*, pages 235–239, 2003.

[22] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, pages 364–377, 2005.

[23] O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In *Compiler Construction*, pages 153–169. Springer, 2003.

[24] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[25] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL. In *OOPSLA*, pages 365–383, 2005.

[26] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *TOSEM*, 14(1):1–41, 2005.

[27] F. Nielson and H. R. Nielson. Interprocedural control flow analysis. In *ESOP*, pages 20–39, 1999.

[28] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android with epicc: an essential step towards holistic security analysis. In *USENIX Security*, 2013.

[29] F. B. Schneider. Enforceable security policies. *TISSEC*, 3(1):30–50, 2000.

[30] O. Shivers. Control flow analysis in scheme. In *PLDI*, pages 164–174. ACM, 1988.

[31] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[32] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, pages 174–184. ACM, 2007.

[33] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387–400. ACM, 2006.

[34] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven pointers-to analysis for java. In *OOPSLA*, pages 59–76. ACM, 2005.

[35] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Software Engineering, IEEE Transactions on*, (1):157–171, 1986.

[36] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, pages 83–95, 2015.

[37] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *CASCON*, page 13, 1999.

[38] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for java. In *ISSTA*, pages 155–165, 2011.

[39] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *ICSE*, 2015.

[40] X. Zheng and R. Rugina. Demand-driven alias analysis for c. *ACM SIGPLAN Notices*, 43(1):197–208, 2008.

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