# A User-Guided Approach to Program Analysis

Ravi Mangal*, Xin Zhang*, Aditya V. Nori†, Mayur Naik*

*Georgia Institute of Technology, USA
{ravi.mangal, xin.zhang, naik}@gatech.edu

†Microsoft Research, UK
adityan@microsoft.com

## ABSTRACT

Program analysis tools often produce undesirable output due to various approximations. We present an approach and a system EUGENE that allows user feedback to guide such approximations towards producing the desired output. We formulate the problem of user-guided program analysis in terms of solving a combination of hard rules and soft rules: hard rules capture soundness while soft rules capture degrees of approximations and preferences of users. Our technique solves the rules using an off-the-shelf solver in a manner that is sound (satisfies all hard rules), optimal (maximally satisfies soft rules), and scales to real-world analyses and programs. We evaluate EUGENE on two different analyses with labeled output on a suite of seven Java programs of size 131–198 KLOC. We also report upon a user study involving nine users who employ EUGENE to guide an information-flow analysis on three Java micro-benchmarks. In our experiments, EUGENE significantly reduces misclassified reports upon providing limited amounts of feedback.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

## Keywords

User feedback, program analysis, report classification

## 1. INTRODUCTION

Program analysis tools often make approximations. These approximations are a necessary evil as the program analysis problem is undecidable in general. There are also several specific factors that drive various assumptions and approximations: program behaviors that the analysis intends to check may be impossible to define precisely (e.g., what constitutes a security vulnerability), computing exact answers may be prohibitively costly (e.g., worst-case exponential in the size of the analyzed program), parts of the analyzed program may be missing or opaque to the analysis (e.g., if the program is a library), and so on. As a result, program analysis tools often produce false positives (or false bugs) and false negatives (or missed bugs), which are absolutely undesirable to users. Users today, however, lack the means to guide such tools towards what they believe to be "interesting" analysis results, and away from "uninteresting" ones.

This paper presents a new approach to *user-guided program analysis*. It shifts decisions about the kind and degree of approximations to apply in an analysis from the *analysis writer* to the *analysis user*. The user conveys such decisions in a natural fashion, by giving feedback about which analysis results she likes or dislikes, and re-running the analysis.

Our approach is a radical departure from existing approaches, allowing users to control both the precision and scalability of the analysis. It offers a different, and potentially more useful, notion of precision—one from the standpoint of the analysis user instead of the analysis writer. It also allows the user to control scalability, as the user's feedback enables tailoring the analysis to the precision needs of the analysis user instead of catering to the broader precision objectives of the analysis writer.

Our approach and tool called EUGENE satisfies three useful goals: (i) *expressiveness*: it is applicable to a variety of analyses, (ii) *automation*: it does not require unreasonable effort by analysis writers or analysis users, and (iii) *precision* and *scalability*: it reports interesting analysis results from a user's perspective and it handles real-world programs. EUGENE achieves each of these goals as described next.

**Expressiveness.** An analysis in EUGENE is expressed as logic inference rules with optional weights (§ 3). In the absence of weights, rules become "hard rules", and the analysis reduces to a conventional one where a solution that satisfies all hard rules is desired. Weighted rules, on the other hand, constitute "soft rules" that generalize a conventional analysis in two ways: they enable to express different degrees of approximation, and they enable to incorporate feedback from the analysis user that may be at odds with the assumptions of the analysis writer. The desired solution of the resulting analysis is one that satisfies all hard rules and maximizes the weight of satisfied soft rules. Such a solution amounts to respecting all indisputable conditions of the analysis writer, while maximizing precision preferences of the analysis user.

**Automation.** EUGENE takes as input analysis rules from the analysis writer, and automatically learns their weights using an *offline learning algorithm* (§ 4.2). EUGENE also requires analysis users to specify which analysis results they like or dislike, and automatically generalizes this feedback

*ESEC/FSE'15*, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786851

```
1   package org.apache.ftpserver;
2   public class RequestHandler {
3       Socket m_controlSocket;
4       FtpRequestImpl m_request;
5       FtpWriter m_writer;
6       BufferedReader m_reader;
7       boolean m_isConnectionClosed;
8       public FtpRequest getRequest() {
9           return m_request;
10      }
11      public void close() {
12          synchronized(this) {
13              if (m_isConnectionClosed)
14                  return;
15              m_isConnectionClosed = true;
16          }
17          m_request.clear();
18          m_request = null;
19          m_writer.close();
20          m_writer = null;
21          m_reader.close();
22          m_reader = null;
23          m_controlSocket.close();
24          m_controlSocket = null;
25      }
26  }
```

**Figure 1: Java code snippet of Apache FTP server.**

**Analysis Relations:**

$\mathsf{next}(p_1, p_2)$ (program point $p_1$ is immediate successor of program point $p_2$)

$\mathsf{parallel}(p_1, p_2)$ (different threads may reach program points $p_1$ and $p_2$ in parallel)

$\mathsf{mayAlias}(p_1, p_2)$ (instructions at program points $p_1$ and $p_2$ may access the same memory location, and constitute a possible datarace)

$\mathsf{guarded}(p_1, p_2)$ (at least one common lock guards program points $p_1$ and $p_2$)

$\mathsf{race}(p_1, p_2)$ (datarace may occur between different threads while executing the instructions at program points $p_1$ and $p_2$)

**Analysis Rules:**

$$\mathsf{parallel}(p_1, p_2) \wedge \mathsf{next}(p_3, p_1) \;\Rightarrow\; \mathsf{parallel}(p_3, p_2) \qquad (1)$$

$$\mathsf{parallel}(p_1, p_2) \;\Rightarrow\; \mathsf{parallel}(p_2, p_1) \qquad (2)$$

$$\left(\begin{array}{rc} \mathsf{parallel}(p_1, p_2) & \wedge \\ \mathsf{mayAlias}(p_1, p_2) & \wedge \\ \neg\mathsf{guarded}(p_1, p_2) & \end{array}\right) \;\Rightarrow\; \mathsf{race}(p_1, p_2) \qquad (3)$$

**Figure 2: Simplified race detection analysis.**

using an *online inference algorithm* (§ 4.1). The analysis rules (hard and soft) together with the feedback from the user (modeled as soft rules) forms a probabilistic constraint system that EUGENE solves efficiently, as described next.

**Precision** and **Scalability.** EUGENE maintains precision by ensuring *integrity* and *optimality* in solving the rules without sacrificing scalability. Integrity (i.e., satisfying hard rules) amounts to respecting indisputable conditions of the analysis. Optimality (i.e., maximally satisfying soft rules) amounts to generalizing user feedback effectively. Together these aspects ensure precision. Satisfying all hard rules and maximizing the weight of satisfied soft rules corresponds to the well-known MaxSAT problem [35]. EUGENE leverages off-the-shelf solvers to solve MaxSAT instances in a manner that is integral, optimal, and scalable.

We demonstrate the precision and scalability of EUGENE on two analyses, namely, datarace detection, and monomorphic call site inference, applied to a suite of seven Java programs of size 131–198 KLOC. We also report upon a user study involving nine users who employ EUGENE to guide an information-flow analysis on three Java micro-benchmarks. In these experiments, EUGENE significantly reduces misclassified reports upon providing limited amounts of feedback.

In summary, our work makes the following contributions:

1. We present a new approach to user-guided program analysis that shifts decisions about approximations in an analysis from the analysis writer to the analysis users, allowing users to tailor its precision and cost to their needs.
2. We formulate our approach in terms of solving a combination of hard rules and soft rules, which enables leveraging off-the-shelf solvers for weight learning and inference that scale without sacrificing integrity or optimality.
3. We show the effectiveness of our approach on diverse analyses applied to a suite of real-world programs. The approach significantly reduces the number of misclassified reports by using only a modest amount of user feedback.

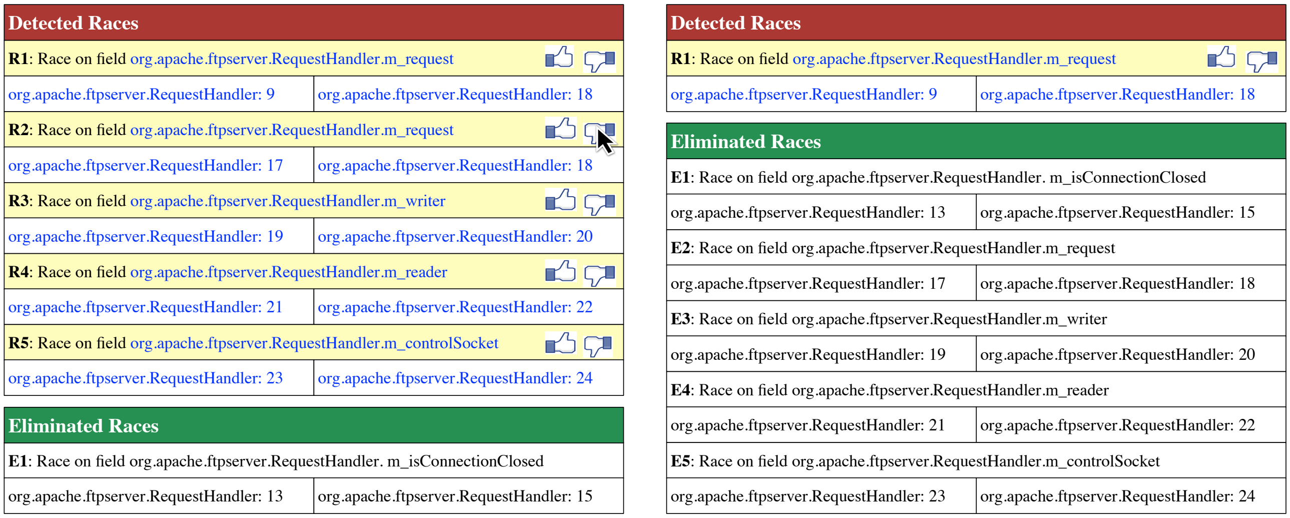
## 2. MOTIVATING EXAMPLE

We illustrate our approach using the example of applying the static race detection tool Chord [30] to a real-world multi-threaded Java program, Apache FTP server [1].

Figure 1 shows a code fragment from the program. The `RequestHandler` class is used to handle client connections and an object of this class is created for every incoming connection to the server. The `close()` method is used to clean up and close an open client connection, while the `getRequest()` method is used to access the `m_request` field. Both these methods can be invoked from various components of the program (not shown), and thus can be simultaneously executed by multiple threads in parallel on the same `RequestHandler` object. To ensure that this parallel execution does not result in any dataraces, the `close()` method uses a boolean flag `m_isConnectionClosed`. If this flag is set, all calls to `close()` return without any further updates. If the flag is not set, then it is first updated to true, followed by execution of the clean-up code (lines 17–24). To avoid dataraces on the flag itself, it is read and updated while holding a lock on the `RequestHandler` object (lines 12–16). All the subsequent code in `close()` is free from dataraces since only the first call to `close()` executes this section. However, note that an actual datarace still exists between the two accesses to field `m_request` on line 9 and line 18.

We motivate our approach by contrasting the goals and capabilities of a *writer of an analysis*, such as the race detection analysis in Chord, with those of a *user of the analysis*, such as a developer of the Apache FTP server.

**The role of the analysis writer.** The designer or writer of a static analysis tool, say Alice, strives to develop an analysis that is precise yet scales to real-world programs, and is widely applicable to a large variety of programs. In the case of Chord, this translates into a race detection analysis that is context-sensitive but path-insensitive. This is a common design choice for balancing precision and scalability of static analyses. The analysis in Chord is expressed using Datalog, a declarative logic programming language, and Figure 2 shows a simplified subset of the logical inference rules used by Chord. The actual analysis implementation uses a larger set of more elaborate rules but the rules shown here suffice for the discussion. These rules are used to produce output relations from input relations, where the input relations express known program facts and output relations express the analysis outcome. These rules express the idioms that the



| Detected Races | | |
|---|---|---|
| **R1**: Race on field org.apache.ftpserver.RequestHandler.m_request | | 👍 👎 |
| org.apache.ftpserver.RequestHandler: 9 | org.apache.ftpserver.RequestHandler: 18 | |
| **R2**: Race on field org.apache.ftpserver.RequestHandler.m_request | | 👍 👎 |
| org.apache.ftpserver.RequestHandler: 17 | org.apache.ftpserver.RequestHandler: 18 | |
| **R3**: Race on field org.apache.ftpserver.RequestHandler.m_writer | | 👍 👎 |
| org.apache.ftpserver.RequestHandler: 19 | org.apache.ftpserver.RequestHandler: 20 | |
| **R4**: Race on field org.apache.ftpserver.RequestHandler.m_reader | | 👍 👎 |
| org.apache.ftpserver.RequestHandler: 21 | org.apache.ftpserver.RequestHandler: 22 | |
| **R5**: Race on field org.apache.ftpserver.RequestHandler.m_controlSocket | | 👍 👎 |
| org.apache.ftpserver.RequestHandler: 23 | org.apache.ftpserver.RequestHandler: 24 | |

| Eliminated Races | |
|---|---|
| **E1**: Race on field org.apache.ftpserver.RequestHandler. m_isConnectionClosed | |
| org.apache.ftpserver.RequestHandler: 13 | org.apache.ftpserver.RequestHandler: 15 |

**(a) Before feedback.**

| Detected Races | | |
|---|---|---|
| **R1**: Race on field org.apache.ftpserver.RequestHandler.m_request | | 👍 👎 |
| org.apache.ftpserver.RequestHandler: 9 | org.apache.ftpserver.RequestHandler: 18 | |

| Eliminated Races | |
|---|---|
| **E1**: Race on field org.apache.ftpserver.RequestHandler. m_isConnectionClosed | |
| org.apache.ftpserver.RequestHandler: 13 | org.apache.ftpserver.RequestHandler: 15 |
| **E2**: Race on field org.apache.ftpserver.RequestHandler.m_request | |
| org.apache.ftpserver.RequestHandler: 17 | org.apache.ftpserver.RequestHandler: 18 |
| **E3**: Race on field org.apache.ftpserver.RequestHandler.m_writer | |
| org.apache.ftpserver.RequestHandler: 19 | org.apache.ftpserver.RequestHandler: 20 |
| **E4**: Race on field org.apache.ftpserver.RequestHandler.m_reader | |
| org.apache.ftpserver.RequestHandler: 21 | org.apache.ftpserver.RequestHandler: 22 |
| **E5**: Race on field org.apache.ftpserver.RequestHandler.m_controlSocket | |
| org.apache.ftpserver.RequestHandler: 23 | org.apache.ftpserver.RequestHandler: 24 |

**(b) After feedback.**

**Figure 3: Race reports produced for Apache FTP server. Each report specifies the field involved in the race, and line numbers of the program points with the racing accesses. The user feedback is to "dislike" report R2.**

analysis writer Alice deems to be the most important for capturing dataraces in Java programs. For example, Rule (1) in Figure 2 conveys that if a pair of program points $(p_1, p_2)$ can execute in parallel, and if program point $p_3$ is an immediate successor of $p_1$, then $(p_3, p_2)$ are also likely to happen in parallel. Rule (2) conveys that the parallel relation is symmetric. Via Rule (3), Alice expresses the idiom that only program points not guarded by a common lock can be potentially racing. In particular, if program points $(p_1, p_2)$ can happen in parallel, can access the same memory location, and are not guarded by any common lock, then there is a potential datarace between $p_1$ and $p_2$.

**The role of the analysis user.** The user of a static analysis tool, say Bob, ideally wants the tool to produce exact (i.e., sound and complete) results on his program. This allows him to spend his time on fixing the bugs in the program instead of classifying the reports generated by the tool as spurious or real. In our example, suppose that Bob runs Chord on the Apache FTP server program in Figure 1. Based on the rules in Figure 2, Chord produces the list of datarace reports shown in Figure 3(a). Reports R1–R5 are identified as potential dataraces in the program, whereas for report E1, Chord detects that the accesses to `m_isConnectionClosed` on lines 13 and 15 are guarded by a common lock, and therefore do not constitute a datarace. Typically, the analysis user Bob is well-acquainted with the program being analyzed, but not with the details of underlying analysis itself. In this case, given his familiarity with the program, it is relatively easy for Bob to conclude that the code from line 17–24 in the body of the `close()` method can never be executed by multiple threads in parallel, and thus reports R2–R5 are spurious.

**The mismatch between analysis writers and users.** The design decisions of the analysis writer Alice have a direct impact on the precision and scalability of the analysis. The datarace analysis in Chord is imprecise for various theoretical and usability reasons.

First, the analysis must scale to large programs. For this reason, it is designed to be path-insensitive and over-approximates the possible thread interleavings. To eliminate spurious reports R2–R5, the analysis would need to only consider feasible thread interleavings by accurately tracking control-flow dependencies across threads. However, such precise analyses do not scale to programs of the size of Apache FTP server, which comprises 130 KLOC.

Scalability concerns aside, relations such as mayAlias are necessarily inexact as the corresponding property is undecidable for Java programs. Chord over-approximates this property by using a context-sensitive but flow-insensitive pointer analysis, resulting in spurious pairs $(p_1, p_2)$ in this relation, which in turn are reported as spurious dataraces.

Third, the analysis writer may lack sufficient information to design a precise analysis, because the program behaviors that the analysis intends to check may be vague or ambiguous. For example, in the case of datarace analysis, real dataraces can be *benign* in that they do not affect the program's correctness [32]. Classifying such reports typically requires knowledge about the program being analyzed.

Fourth, the program specification can be incomplete. For instance, the race in report R1 above could be harmful but impossible to trigger due to timing reasons extrinsic to Apache FTP server, such as the hardware environment.

In short, while the analysis writer Alice can influence the design of the analysis, she cannot foresee every usage scenario or program-specific tweaks that might improve the analysis. Conversely, analysis user Bob is acquainted with the program under analysis, and can classify the analysis reports as spurious or real. But he lacks the tools or expertise to suppress the spurious bugs by modifying the underlying analysis based on his intuition and program knowledge.

**Closing the gap between analysis writers and users.** Our user-guided approach aims to empower the analysis user Bob to adjust the underlying analysis as per his demands without involving the analysis writer Alice. Our system, EUGENE, achieves this by automatically incorporating user feedback into the analysis. The user provides feedback in a natural fashion, by "liking" or "disliking" a subset of the analysis reports, and re-running the analysis. For example,

$$\begin{array}{rl rl}
\text{(relation)} & r \in \mathbf{R} & \text{(argument)} & a \in \mathbf{A} = \mathbf{V} \cup \mathbf{C} \\
\text{(constant)} & c \in \mathbf{C} & \text{(fact)} & t \in \mathbf{T} = \mathbf{R} \times \mathbf{A}^* \\
\text{(variable)} & v \in \mathbf{V} & \text{(ground fact)} & g \in \mathbf{G} = \mathbf{R} \times \mathbf{C}^* \\
\text{(valuation)} & \sigma \in \mathbf{V} \to \mathbf{C} & \text{(weight)} & w \in \mathbb{R}+ = (0, \infty]
\end{array}$$

$$\begin{array}{rlll}
\text{(hard rules)} & H ::= \{h_1, ..., h_n\}, & h ::= \bigwedge_{i=1}^{n} t_i \Rightarrow \bigvee_{i=1}^{m} t'_i \\
\text{(soft rules)} & S ::= \{s_1, ..., s_n\}, & s ::= (h, w)
\end{array}$$

$$\begin{array}{rl}
\text{(probabilistic analysis)} & C ::= (H, S) \\
\text{(analysis input, output)} & P, Q \subseteq \mathbf{G}
\end{array}$$

**Figure 4: Syntax of a probabilistic analysis.**

when presented with the datarace reports in Figure 3(a), Bob might start inspecting from the first report. This report is valid and Bob might choose to either like or ignore this report. Liking a report conveys that Bob accepts the reported bug as a real one and would like the analysis to generate more similar reports, thereby reinforcing the behavior of the underlying analysis that led to the generation of this report. However, suppose that Bob ignores the first report, but indicates that he dislikes the second report by clicking on the corresponding icon. Re-running Chord after providing this feedback produces the reports shown in Figure 3(b). While the true report R1 is generated in this run as well, all the remaining spurious reports are eliminated. This highlights a key strength of our approach: Eugene not only incorporates user feedback, but it also generalizes the feedback to other similar results of the analysis. Reports R2–R5 are correlated and are spurious for the same root cause: the code from line 17–24 in the body of the `close`() method can never be executed by multiple threads in parallel. Bob's feedback on report R2 conveys to the underlying analysis that lines 17 and 18 cannot execute in parallel. Eugene is able to generalize this feedback automatically and conclude that none of the lines from 17–24 can execute in parallel.

In the following section, we describe the underlying details of Eugene that allow it to incorporate user feedback and generalize it automatically to other reports.

## 3. ANALYSIS SPECIFICATION

Eugene uses a constraint-based approach wherein analyses are written in a declarative constraint language. Constraint languages have been widely adopted to specify a broad range of analyses. The declarative nature of such languages allows analysis writers to focus on the high-level analysis logic while delegating low-level implementation details to off-the-shelf constraint solvers. In particular, Datalog, a logic programming language, is widely used in such approaches. Datalog has been shown to suffice for expressing a variety of analyses, including pointer and call-graph analyses [7,40,41,44], concurrency analyses [30,31], security analyses [11,27], and reflection analysis [25].

Existing constraint-based approaches allow specifying only *hard rules* where an acceptable solution is one that satisfies all the rules. However, this is insufficient for incorporating feedback from the analysis user that may be at odds with the assumptions of the analysis writer. To enable the flexibility of having conflicting constraints, it is necessary to allow *soft rules* that an acceptable solution can violate. Our user-guided approach is based on such a constraint language that extends Datalog rules with weights. We refer to analyses specified in this extended language as *probabilistic analyses*.

**Probabilistic analysis syntax.** A *probabilistic analysis* $C$, defined in Figure 4, consists of a set of hard rules $H$ and a set of soft rules $S$. A hard rule $h \in H$ is an inference rule $A \Rightarrow B$, where $A$ is a conjunction of facts and $B$ is a disjunction of facts. An analysis *fact* $t$ comprises a relation name and a tuple of arguments, which include variables and constants; a fact is a *ground fact* $g$ when all arguments are constants. Our setting subsumes logical analyses written in Datalog, where all rules are Horn rules, i.e., rules with at most one disjunct on the right hand side ($|B| = 1$).

$$\begin{array}{rrcl}
\text{(ground clause)} & \rho & ::= & \bigvee_{i=1}^{n} \neg g_i \vee \bigvee_{i=1}^{m} g'_i \\
\text{(hard clauses)} & \phi & ::= & \bigwedge_{i=1}^{n} \rho_i \\
\text{(soft clauses)} & \psi & ::= & \bigwedge_{i=1}^{n} (\rho_i, w_i)
\end{array}$$

$$\mathtt{MaxSAT}(\phi, \bigwedge_{i=1}^{n} (\rho_i, w_i)) =$$

$$\begin{cases} \textsc{unsat} & \text{if } \nexists Q : Q \models \phi \\ Q \text{ such that} \left[ \begin{array}{l} Q \models \phi \text{ and} \\ \Sigma_{i=1}^{n} \{ w_i \mid Q \models \rho_i \} \text{ is maximized} \end{array} \right] & \text{otherwise} \end{cases}$$

$$\begin{array}{lll}
Q \models \bigwedge_{i=1}^{n} \rho_i & \text{iff } \forall i : Q \models \rho_i \\
Q \models \bigvee_{i=1}^{n} \neg g_i \vee \bigvee_{i=1}^{m} g'_i & \text{iff } \exists i : g_i \notin Q \text{ or } \exists i : g'_i \in Q
\end{array}$$

**(a) Syntax and semantics of a MaxSAT formula.**

$$\begin{array}{rcl}
\llbracket (H, S) \rrbracket & = & (\llbracket H \rrbracket, \llbracket S \rrbracket) \\
\llbracket \{h_1, ..., h_n\} \rrbracket & = & \bigwedge_{i=1}^{n} \llbracket h_i \rrbracket \\
\llbracket \{s_1, ..., s_n\} \rrbracket & = & \bigwedge_{i=1}^{n} \llbracket s_i \rrbracket \\
\llbracket h \rrbracket & = & \bigwedge_{\sigma} \llbracket h \rrbracket_{\sigma} \\
\llbracket (h, w) \rrbracket & = & \bigwedge_{\sigma} (\llbracket h \rrbracket_{\sigma}, w) \\
\llbracket \bigwedge_{i=1}^{n} t_i \Rightarrow \bigvee_{i=1}^{m} t'_i \rrbracket_{\sigma} & = & (\bigvee_{i=1}^{n} \neg \llbracket t_i \rrbracket_{\sigma} \vee \bigvee_{i=1}^{m} \llbracket t'_i \rrbracket_{\sigma}) \\
\llbracket r(a_1, ..., a_n) \rrbracket_{\sigma} & = & r(\llbracket a_1 \rrbracket_{\sigma}, ..., \llbracket a_n \rrbracket_{\sigma}) \\
\llbracket v \rrbracket_{\sigma} & = & \sigma(v) \\
\llbracket c \rrbracket_{\sigma} & = & c
\end{array}$$

**(b) Compiling a probabilistic analysis to MaxSAT.**

**Figure 5: Semantics of a probabilistic analysis.**

A soft rule $s \in S$ is a hard rule along with a positive weight $w$. A weight has a natural probabilistic interpretation, where the confidence associated with a soft rule increases with the weight. For the precise semantics of weights, the reader is referred to [10]. In the absence of soft rules, the analysis reduces to a *logical analysis* consisting of only hard rules.

The analysis input (a program $P$) and the analysis output (a result $Q$) are represented as sets of ground facts.

**Probabilistic analysis semantics.** We define the semantics of a probabilistic analysis in terms of an optimization extension of the Boolean satisfiability (SAT) problem, called the weighted partial maximum satisfiability (MaxSAT) problem [35], shown as procedure `MaxSAT` in Figure 5(a). This procedure takes as input a MaxSAT formula comprising a set of hard clauses $\phi$ and a set of soft clauses $\psi$. These are counterparts of hard rules and soft rules in a probabilistic analysis. The key difference is that all facts in each (hard or soft) MaxSAT clause $\rho$ are grounded, whereas analysis rules can contain ungrounded facts. The `MaxSAT` procedure views each unique ground fact in the input MaxSAT formula as a separate Boolean variable. The procedure returns either: (1) unsat, if no assignment of truth values to the Boolean variables satisfies the set of hard clauses $\phi$, or (2) a solution $Q$, denoting the assignment "$\lambda g.(g \in Q)$ ? true : false", i.e., it sets variables corresponding to ground facts contained in $Q$ to true, and the rest to false. The solution $Q$ not only satisfies all hard clauses in $\phi$ but it also maximizes the sum of the weights of satisfied soft clauses in $\psi$. Note that $Q$ is not necessarily unique; two solutions $Q_1$ and $Q_2$ are *equivalent* if $\mathtt{Weight}(Q_1, \psi) = \mathtt{Weight}(Q_2, \psi)$.

The compilation of a probabilistic analysis to a MaxSAT formula is shown in Figure 5(b). The compilation proce-

**Input facts:**

$\mathsf{next}(18,17)$ $\mathsf{mayAlias}(18,17)$ $\mathsf{guarded}(13,13)$
$\mathsf{next}(19,18)$ $\mathsf{mayAlias}(20,19)$ $\mathsf{guarded}(15,15)$
$\mathsf{next}(20,19)$ $\mathsf{mayAlias}(22,21)$ $\mathsf{guarded}(15,13)$ ...

**MaxSAT formula:**

$$\begin{array}{ll}
w_1 : (\neg\mathsf{parallel}(17,17) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,17)) & \wedge \\
\quad\ (\neg\mathsf{parallel}(18,17) \vee \mathsf{parallel}(17,18)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(17,18) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,18)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(18,18) \vee \neg\mathsf{next}(19,18) \vee \mathsf{parallel}(19,18)) & \wedge \\
\quad\ (\neg\mathsf{parallel}(19,18) \vee \mathsf{parallel}(18,19)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(18,19) \vee \neg\mathsf{next}(19,18) \vee \mathsf{parallel}(19,19)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(19,19) \vee \neg\mathsf{next}(20,19) \vee \mathsf{parallel}(20,19)) & \wedge \\
\quad \left(\begin{array}{cccc} \neg\mathsf{parallel}(18,17) & \vee & \neg\mathsf{mayAlias}(18,17) & \vee \\ \mathsf{guarded}(18,17) & \vee & \mathsf{race}(18,17) & \end{array}\right) & \wedge \\
\quad \left(\begin{array}{cccc} \neg\mathsf{parallel}(20,19) & \vee & \neg\mathsf{mayAlias}(20,19) & \vee \\ \mathsf{guarded}(20,19) & \vee & \mathsf{race}(20,19) & \end{array}\right) & \wedge \\
\boxed{w_2 : \neg\mathsf{race}(18,17)} & \wedge \ \ldots
\end{array}$$

**Output facts (before feedback):**

$\mathsf{parallel}(18,9)$ $\mathsf{parallel}(20,19)$ $\mathsf{race}(18,9)$ $\mathsf{race}(20,19)$ ...
$\mathsf{parallel}(18,17)$ $\mathsf{parallel}(22,21)$ $\mathsf{race}(18,17)$ $\mathsf{race}(22,21)$

**Output facts (after feedback):**

$\mathsf{parallel}(18,9)$ $\mathsf{race}(18,9)$ ...

**Figure 6: Probabilistic analysis example.**

dure grounds each analysis rule into a set of corresponding MaxSAT clauses. In particular, the conversion $[\![h]\!] = \bigwedge_\sigma [\![h]\!]_\sigma$ grounds analysis rule $h$ by enumerating all possible groundings $\sigma$ of variables to constants, producing a MaxSAT clause for each unique valuation to the variables in $h$.

There are a number of solvers [8,14,26,33,34,37] that efficiently compile a probabilistic analysis down into a MaxSAT formula, and solve the corresponding formula to produce the desired output that satisfies all the hard rules and maximizes the weight of the satisfied soft rules. EUGENE treats the underlying solver as a black-box, and can use any of these solvers to solve the constraints of a probabilistic analysis.

**Example.** Equipped with the concept of probabilistic analysis, we can now describe how EUGENE works on the race detection example from § 2. Figure 6 shows a subset of the input and output facts as well as a snippet of the MaxSAT formula constructed for the example. The input facts are derived from the analyzed program (Apache FTP server) and comprise the $\mathsf{next}$, $\mathsf{mayAlias}$, and $\mathsf{guarded}$ relations. In all these relations, the domain of program points is represented by the corresponding line number in the code. Note that all the analysis rules expressed in Figure 2 are hard rules since existing tools like Chord do not accept soft rules. However, we assume that when this analysis is fed to EUGENE, rule (1) is specified to be soft by analysis writer Alice, which captures the fact that the $\mathsf{parallel}$ relation is imprecise. EUGENE automatically learns the weight of this rule to be $w_1$ from training data (see § 4.2 for details). Given these input facts and rules, the MaxSAT problem to be solved is generated by grounding the analysis rules, and a snippet of the constructed MaxSAT formula is shown in Figure 6. Ignoring the clause enclosed in the box, solving this MaxSAT formula (without the boxed clause) yields output facts, a subset of which is shown under "Output facts (before feedback)" in Figure 6. The output includes multiple spurious races like $\mathsf{race}(18,17)$, $\mathsf{race}(20,19)$, and $\mathsf{race}(22,21)$.

As described in § 2, when analysis user Bob provides feedback that $\mathsf{race}(18,17)$ is spurious, EUGENE suppresses all spurious races while retaining the real race $\mathsf{race}(18,9)$. EUGENE achieves this by incorporating the user feedback itself as a soft rule, represented by the boxed clause $\neg\mathsf{race}(18,17)$ in Figure 6. The weight for such user feedback is also learned during the training phase. Assuming the weight $w_2$ of the feedback clause is higher than the weight $w_1$ of rule (1)—a reasonable choice that emphasizes Bob's preferences over Alice's assumptions—the MaxSAT semantics ensures that the solver prefers violating rule (1) over violating the feedback clause. When the MaxSAT formula (with the boxed clause) in Figure 6 is then fed to the solver, the output solution violates the clause $w_1 : (\neg\mathsf{parallel}(17,17) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,17))$ and does not produce facts $\mathsf{parallel}(18,17)$ and $\mathsf{race}(18,17)$ in the output. Further, all the facts that are dependent on $\mathsf{parallel}(18,17)$ are not produced either.[1] This implies that facts like $\mathsf{parallel}(19,18)$, $\mathsf{parallel}(20,19)$, $\mathsf{parallel}(22,21)$ are not produced, and therefore $\mathsf{race}(20,19)$ and $\mathsf{race}(22,21)$ are also suppressed. Thus, EUGENE is able to generalize based on user feedback. The degree of generalization depends on the quality of the weights assigned or learned for the soft rules.

# 4. THE EUGENE SYSTEM

This section describes our system EUGENE for user-guided program analysis. Its workflow, shown in Figure 7, comprises an online inference stage and an offline learning stage.

In the online stage, EUGENE takes the probabilistic analysis specification together with a program $P$ that an analysis user Bob wishes to analyze. The inference engine, described in § 4.1, uses these inputs to produce the analysis output $Q$. Further, the online stage allows Bob to provide feedback on the produced output $Q$. In particular, Bob can indicate the output queries he likes ($Q_\mathsf{L}$) or dislikes ($Q_\mathsf{D}$), and invoke the inference engine with $Q_\mathsf{L}$ and $Q_\mathsf{D}$ as additional inputs. The inference engine incorporates Bob's feedback as additional soft rules in the probabilistic analysis specification used for producing the new result $Q$. This interaction continues until Bob is satisfied with the analysis output.

The accuracy of the produced results in the online stage is sensitive to the weights assigned to the soft rules. Manually assigning weights is not only inefficient, but in most cases it is also infeasible since weight assignment needs analysis of data. Therefore, EUGENE provides an offline stage that automatically learns the weights of soft rules in the probabilistic analysis specification. In the offline stage, EUGENE takes a logical analysis specification from analysis writer Alice and training data in the form of a set of input programs and desired analysis output on these programs. These inputs are fed to the learning engine described in § 4.2. The logical analysis specification includes hard rules as well as rules marked as soft whose weights need to be learnt. The learning engine infers these weights to produce the probabilistic analysis specification. The learning engine ensures that the learnt weights maximize the likelihood of the training data with respect to the probabilistic analysis specification.

## 4.1 Online Component of EUGENE: Inference

Algorithm 1 describes the online component `Inference` of EUGENE. `Inference` takes as input, a probabilistic analysis $(H, S)$ (with learnt weights), and the program $P$ to be analyzed. First, in line 6, the algorithm augments the hard and soft rules $(H, S)$ of the analysis with the inputs $P$, $Q_\mathsf{L}$, $Q_\mathsf{D}$ to

[1] This is due to implicit soft rules that negate each output relation, such as $w_0 : \neg\mathsf{parallel}(p_1, p_2)$ where $w_0 < w_1$, in order to obtain the least solution.

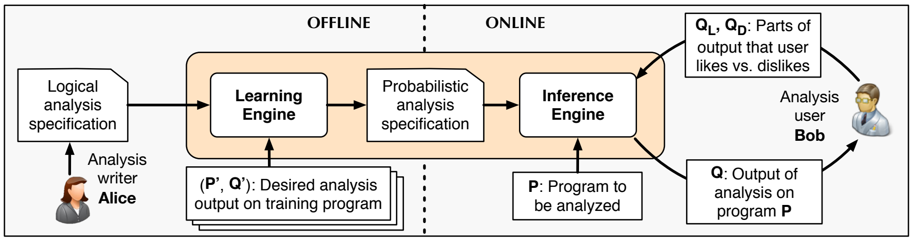

OFFLINE
ONLINE
Logical analysis specification
Analysis writer Alice
Learning Engine
(P', Q'): Desired analysis output on training program
Probabilistic analysis specification
Inference Engine
P: Program to be analyzed
Q_L, Q_D: Parts of output that user likes vs. dislikes
Analysis user Bob
Q: Output of analysis on program P

**Figure 7: Workflow of the Eugene system for user-guided program analysis.**

**Algorithm 1** `Inference`: Online component of EUGENE.

```
1:  PARAM $(w_l, w_d)$: Weights of liked and disliked queries.
2:  INPUT $C = (H, S)$: Probabilistic analysis.
3:  INPUT $P$: Program to analyze.
4:  OUTPUT $Q$: Final output of user-guided analysis.
5:  $Q_{\mathsf{L}} := \emptyset$;  $Q_{\mathsf{D}} := \emptyset$
6:  $H' := H \cup P$
7:  repeat
8:      $S' := S \cup \{(g, w_l) \mid g \in Q_{\mathsf{L}}\} \cup \{(\neg g, w_d) \mid g \in Q_{\mathsf{D}}\}$
9:      $Q := solve(H', S')$
10:     $Q_{\mathsf{L}} :=$ PositiveUserFeedback($Q$)
11:     $Q_{\mathsf{D}} :=$ NegativeUserFeedback($Q$)
12: until $Q_{\mathsf{L}} \cup Q_{\mathsf{D}} = \emptyset$
```

**Algorithm 2** `Learning`: Offline component of EUGENE.

```
1:  PARAM $\alpha$: rate of change of weight of soft rules.
2:  INPUT $C = (H, S)$: Initial probabilistic analysis.
3:  INPUT $Q$: Desired analysis output.
4:  OUTPUT $C'$: Probabilistic analysis with learnt weights.
5:  $S' := \{(h, w') \mid \exists w\colon (h, w) \in S$ and $w' = \log(n_1/n_2)\}$
6:  where $n_1 = |$Groundings$(h, Q)|$, $n_2 = |$Violations$(h, Q)|$.
7:  repeat
8:      $C' := (H, S')$
9:      $Q' :=$ Inference$(C', \emptyset)$
10:     $S := S'$
11:     $S' := \{ \ (h, w') \mid \exists w\colon (h, w) \in S$ and
12:                $w' = w + \alpha * (n_1 - n_2)\}$
13:     where $n_1 = |$Violations$(h, Q')|$, $n_2 = |$Violations$(h, Q \ )|$.
14: until $S' = S$
15: Violations$(h, Q) = \{ \ [\![h]\!]_\sigma \mid Q \not\models [\![h]\!]_\sigma \ \}$
16: Groundings$(h, Q) = \{ \ [\![h]\!]_\sigma \mid Q \models [\![h]\!]_\sigma \ \}$
```

the analysis, to obtain an extended set of rules $(H', S')$ (lines 6 and 8). Notice that the user feedback $Q_{\mathsf{L}}$ (liked queries) and $Q_{\mathsf{D}}$ (disliked queries) are incorporated as soft rules in the extended rule set. Each liked query feedback is assigned the fixed weight $w_l$, while each disliked query feedback is assigned weight $w_d$ (line 8). Weights $w_l$ and $w_d$ are learnt in the offline stage and fed as parameters to Algorithm 1. Instead of using fixed weights for the user feedback, two other options are: (a) treating user feedback as hard rules, and (b) allowing a different weight for each query feedback. Option (a) does not account for users being wrong, leaving no room for the inference engine to ignore the feedback if necessary. Option (b) is too fine-grained, requiring learning separate weights for each query. We therefore take a middle ground between these two extremes.

Next, in line 9, the algorithm invokes a weighted constraints solver [26] with the extended set of rules. Note that EUGENE treats the solver as a black-box and any suitable solver suffices. The solver produces a solution $Q$ that satisfies all the hard rules in the extended set, while maximizing the weight of satisfied soft rules. The solution $Q$ is then presented to Bob who can give his feedback by liking or disliking the queries (lines 10–11). The sets of liked and disliked queries, $Q_{\mathsf{L}}$ and $Q_{\mathsf{D}}$, are used to further augment the hard and soft rules $(H, S)$ of the analysis. This loop (lines 7–12) continues until no further feedback is provided by Bob.

## 4.2 Offline Component of EUGENE: Learning

Algorithm 2 describes the offline component `Learning` of EUGENE. It is an adaptation of [38] to our application. `Learning` takes a probabilistic analysis $C = (H, S)$ with arbitrary weights, a set of programs $\mathcal{P}$ and the desired analysis output $Q$ as input, and outputs a probabilistic analysis $C'$ with learnt weights. Without loss of generality, we assume that $\mathcal{P}$ is encoded as a set of hard clauses and is part of $H$.

As a first step, in line 5, `Learning` assigns initial weights to all the soft rules. The initial weight of a rule $h \in S$ is computed as a log of the ratio of the number of groundings of $h$ satisfied by the desired output $Q$ to the number of violations of $h$ by $Q$ (lines 5–6). In other words, the initial weight captures the log odds of a rule being true in the training data. Note that, in the case `Violations`$(h, Q) = 0$, it is substituted by a suitably small value [38].

Next, in line 9, the probabilistic analysis $C'$ (defined in line 8) with the initialized weights is fed to the `Inference` procedure described in Algorithm 1. This produces a solution $Q'$ that is integral and optimal for the probabilistic analysis $C'$. The solution $Q'$ is then used to update the weights of the soft rules. The weights are updated according to the formulae in lines 10-13. The basic intuition for these update rules is as follows: weights learnt by the learning algorithm must be such that the output solution of the `Inference` algorithm for the training program is as close to the desired output $Q$ as possible. Towards that end, if the current output $Q'$ produces more violations for a rule than the desired output, it implies that the rule needs to be strengthened and its weight should be increased. On the other hand, if the current output $Q'$ produces fewer violations for a rule then $Q$, the rule needs to be weakened and its weight should be reduced. The formula in the algorithm has exactly the same effect as described here. Moreover, the rate of change of weights can be controlled by an input parameter $\alpha$. The learning process continues iteratively until the learnt weights do not change. In practice, the learning process can be terminated after a fixed number of iterations, or when the difference in weights between successive iterations does not change significantly.

# 5. EMPIRICAL EVALUATION

We implemented EUGENE atop Chord [29], an extensible program analysis framework for Java bytecode that supports writing analyses in Datalog. In our evaluation, we investigate the following research questions:

- **RQ1:** Does using EUGENE improve analysis precision for practical analyses applied to real-world programs? How much feedback is needed for the same, and how does the amount of provided feedback affect the precision?
- **RQ2:** Does EUGENE scale to large programs? Does the amount of feedback influence the scalability?
- **RQ3:** How feasible is it for users to inspect analysis output and provide useful feedback to EUGENE?

## 5.1 Experimental Setup

We performed two different studies with EUGENE: a control study and a user study.

First, to evaluate the precision and scalability of EUGENE, we performed a *control study* using two realistic analyses expressed in Datalog applied to seven Java benchmark programs. The goal of this study is to thoroughly investigate the performance of EUGENE in realistic scenarios and with varying amounts of feedback. To practically enable the evaluation of EUGENE over a large number of a data-points in the $(benchmark, analysis, \#feedback)$ space, this study uses a more precise analysis, instead of a human user, as an oracle for generating the feedback to be provided. This study helps us evaluate **RQ1** and **RQ2**.

Second, to evaluate the practical usability of EUGENE when human analysis users are in the loop, we conducted a *user study* with nine users who employed EUGENE to guide an information-flow analysis on three benchmark programs. In contrast to the first study, the human users provide the feedback in this case. This study helps us evaluate **RQ3**.

All experiments were run using Oracle HotSpot JVM 1.6.0 on a Linux server with 64GB RAM and 3.0GHz processors.

**Clients.** Our two analyses in the first study (Table 1) are datarace detection (*datarace*) and monomorphic call site inference (*polysite*), while we use an information-flow (*infoflow*) analysis for the user study. Each of these analyses is sound, and composed of other analyses written in Datalog. For example, *datarace* includes a thread-escape analysis and a may-happen-in-parallel analysis, while *polysite* and *infoflow* include a pointer analysis and a call-graph analysis. The pointer analysis used here is a flow/context-insensitive, field-sensitive, Andersen-style analysis using allocation site heap abstraction [20]. The *datarace* analysis is from [30], while the *polysite* analysis has been used in previous works [22, 42, 45] to evaluate pointer analyses. The *infoflow* analysis only tracks explicit information flow similar to the analysis described in [23]. For scalability reasons, all these analyses are context-, object-, and flow-insensitive, which is the main source of false positives reported by them.

**Benchmarks.** The benchmarks for the first study (upper seven rows of Table 2) are 131–198 KLOC in size, and include programs from the DaCapo suite [5] (`antlr`, `avrora`, `luindex`, `lusearch`) and from past works that address our two analysis problems.

The benchmarks for the user study (bottom three rows of Table 2) are 0.6–4.2 KLOC in size, and are drawn from Securibench Micro [3], a micro-benchmark suite designed to exercise different parts of a static information-flow analyzer.

**Methodology.** We describe the methodology for the offline (learning) and online (inference) stages of EUGENE.

***Offline stage.*** We first converted the above three logical analyses into probabilistic analyses using the offline training stage of EUGENE. To avoid selection bias, we used a set of small benchmarks for training instead of those in Table 2. Specifically, we used `elevator` and `tsp` (100 KLOC each) from [2]. While the training benchmarks are smaller and fewer than the testing benchmarks, they are sizable, realistic, and disjoint from those in the evaluation, demonstrating the practicality of our training component. Besides the sample programs, the training component of EUGENE also requires the expected output of the analyses on these sample programs. Since the main source of false positives in our analyses is the lack of context- and object-sensitivity, we used context- and object-sensitive versions of these analyses as oracles for generating the expected output. Specifically, we used $k$-object-sensitive versions [28] with cloning depth $k = 4$. Note that these oracle analyses used for generating the training data comprise their own approximations (for example, flow-insensitivity), and thus do not produce the absolute ground truth. Using better training data would only imply that the weights learnt by EUGENE are more reflective of the ground truth, leading to more precise results.

**Table 1: Statistics of our probabilistic analyses.**

| | rules | input relations | output relations |
|---|---|---|---|
| *datarace* | 30 | 18 | 18 |
| *polysite* | 76 | 50 | 42 |
| *infoflow* | 76 | 52 | 42 |

***Online stage.*** We describe the methodology for the online stage separately for the control study and the user study.

**Control study methodology.** To perform the control study, we started by running the inference stage of EUGENE on our probabilistic analyses (*datarace* and *polysite*) with no feedback to generate the initial set of reports for each benchmark. Next, we simulated the process of providing feedback by: (i) randomly choosing a subset of the initial set of reports, (ii) classifying each of the reports in the chosen subset as spurious or real, and (iii) re-running the inference stage of EUGENE on the probabilistic analyses with the labeled reports in the chosen subset as feedback. To classify the reports as spurious or real, we used the results of $k$-object-sensitive versions of our client analyses as ground truth. In other words, if a report contained in the chosen subset is also generated by the precise version of the analysis, it is classified as a real report, otherwise it is labeled as a spurious report. For each $(benchmark, analysis)$ pair, we generated random subsets that contain 5%, 10%, 15%, and 20% of the initial reports. This allows us to study the effect of varying amounts of feedback on EUGENE's performance. Additionally, EUGENE can be sensitive to not just the amount of feedback, but also to the actual reports chosen for feedback. To discount this effect, for a given $(benchmark, analysis)$ pair, and a given feedback subset size, we ran EUGENE thrice using different random subsets of the given size in each run. Randomly choosing feedback ensures that we conservatively estimate the performance of EUGENE. Finally, we evaluated the quality of the inference by comparing the output of EUGENE with the output generated by the $k$-object-sensitive versions of our analyses with $k = 4$.

**User study methodology.** For the user study, we engaged nine users, all graduate students in computer science, to run EUGENE on *infoflow* analysis. Each user was assigned two benchmarks from the set of {`secbench1`, `secbench2`, `secbench3`}, such that each of these benchmarks was assigned to six users in total. The users interacted with EUGENE by first running it without any feedback so as to produce

Table 2: Benchmark statistics. Columns "total" and "app" are with and without JDK library code.

| | brief description | # classes | | # methods | | bytecode (KB) | | source (KLOC) | |
|---|---|---|---|---|---|---|---|---|---|
| | | app | total | app | total | app | total | app | total |
| `antlr` | parser/translator generator | 111 | 350 | 1,150 | 2,370 | 128 | 186 | 29 | 131 |
| `avrora` | microcontroller simulator/analyzer | 1,158 | 1,544 | 4,234 | 6,247 | 222 | 325 | 64 | 193 |
| `ftp` | Apache FTP server | 93 | 414 | 471 | 2,206 | 29 | 118 | 13 | 130 |
| `hedc` | web crawler from ETH | 44 | 353 | 230 | 2,134 | 16 | 140 | 6 | 153 |
| `luindex` | document indexing and search tool | 206 | 619 | 1,390 | 3,732 | 102 | 235 | 39 | 190 |
| `lusearch` | text indexing and search tool | 219 | 640 | 1,399 | 3,923 | 94 | 250 | 40 | 198 |
| `weblech` | website download/mirror tool | 11 | 576 | 78 | 3,326 | 6 | 208 | 12 | 194 |
| `secbench1` | securibench micro 1 | 4 | 5 | 10 | 13 | 0.3 | 0.3 | 0.08 | 0.6 |
| `secbench2` | securibench micro 2 | 3 | 4 | 9 | 12 | 0.2 | 0.2 | 0.07 | 0.6 |
| `secbench3` | securibench micro 3 | 2 | 17 | 4 | 46 | 0.3 | 1.25 | 0.06 | 4.2 |

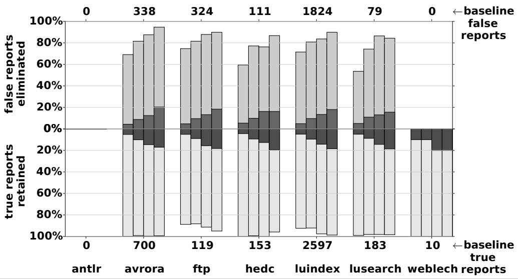


**Figure 8: Results of Eugene on *datarace* analysis.**

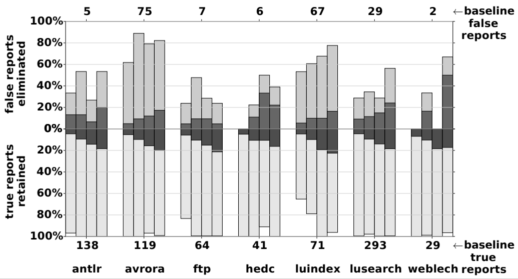


**Figure 9: Results of Eugene on *polysite* analysis.**

the initial set of reports. The users then analyzed these produced reports, and were asked to provide any eight reports with their corresponding label (spurious or real) as feedback. Also, for each benchmark, we recorded the time spent by each user in analyzing the reports and generating the feedback. Next, Eugene was run with the provided feedback, and the produced output was compared with manually generated ground truth for each of the benchmarks.

We next describe the results of evaluating Eugene's precision (**RQ1**), scalability (**RQ2**), and usability (**RQ3**).

## 5.2 Precision of Eugene

The analysis results of our control study under varying amounts of feedback are shown in Figures 8 and 9. In these figures, "baseline false reports" and "baseline true reports" are the number of false and true reports produced when Eugene is run without any feedback. The light colored bars above and below the x-axis indicate the % of false reports eliminated and the % of true reports retained, respectively, when the % of feedback indicated by the corresponding dark colored bars is provided. For each benchmark, the feedback percentages increase from left to right, i.e., 5% to 20%. Ideally, we want all the false reports to be eliminated and all the true reports to be retained, which would be indicated by the light color bars extending to 100% on both sides.

Even without any feedback, our probabilistic analyses are already fairly precise and sophisticated, and eliminate all except the non-trivial false reports. Despite this, Eugene helps eliminate a significant number of such hard-to-refute reports. On average 70% of the false reports are eliminated across all our experiments with 20% feedback. Likewise importantly, on average 98% of the true reports are retained when 20% feedback is provided. Also, note that with 5% feedback the percentage of false reports eliminated falls to 44% on average, while that of true reports retained is 94%. A finer-grained look at the results for individual benchmarks and analyses reveals that in many cases, increasing feedback only leads to modest gains.

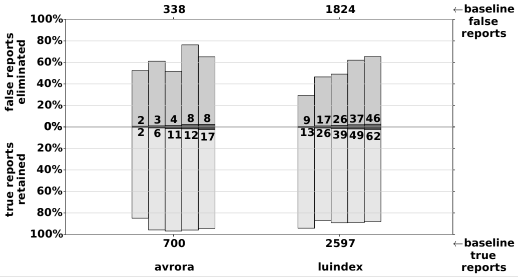


**Figure 10: Results of Eugene on *datarace* analysis with feedback (0.5%,1%,1.5%,2%,2.5%).**

We next discuss the precision of Eugene for each of our probabilistic analyses. For *datarace*, with 20% feedback, an average of 89% of the false reports are eliminated while an average of 98% of the true reports are retained. Further, with 5% feedback the averages are 66% for false reports eliminated and 97% for true reports retained. Although the precision of Eugene increases with more feedback in this case, the gains are relatively modest. Note that given the large number of initial reports generated for `luindex` and `avrora` (4421 and 1038 respectively), it is somewhat impractical to expect analysis users to provide up to 20% feedback. Consequently, we re-run Eugene for these benchmarks with 0.5%, 1%, 1.5%, 2% and 2.5% feedback. The results are shown in Figure 10. Interestingly, we observe that for `luindex`, with only 2% feedback on the false reports and 1.9% feedback on true reports, Eugene eliminates 62% of false reports and retains 89% of the true reports. Similarly for `avrora`, with only 2.3% feedback on the false reports and 1.8% feedback on true reports, Eugene eliminates 76% of false reports and retains 96% of the true reports. These numbers indicate that, for the *datarace* client, Eugene is able to generalize even with a very limited amount of user feedback.

For *polysite*, with 20% feedback, an average of 57% of the false reports are eliminated and 99% of the true reports are retained, while with 5% feedback, 29% of the false reports are eliminated and 92% of the true reports are re-

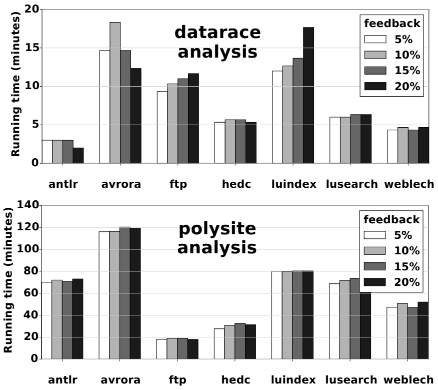


**Figure 11: Running time of** Eugene.

tained. There are two important things to notice here. First, the number of eliminated false reports does not always grow monotonically with more feedback. The reason is that Eugene is sensitive to the reports chosen for feedback, but in each run, we randomly choose the reports to provide feedback on. Though the precision numbers here are averaged over three runs for a given feedback amount, the randomness in choosing feedback still seeps into our results. Second, Eugene tends to do a better job at generalizing the feedback for the larger benchmarks compared to the smaller ones. We suspect the primary reason for this is the fact that smaller benchmarks tend to have a higher percentage of bugs with unique root causes, and thereby a smaller number of bugs are attributable to each unique cause. Consequently, the scope for generalization of the user feedback is reduced.

Answer to **RQ1**: Eugene significantly reduces false reports with only modest feedback, while retaining the vast majority of true reports. Though increasing feedback leads to more precise results in general, for many cases, the gain in precision due to additional feedback is modest.

## 5.3 Scalability of Eugene

The performance of Eugene for our control study, in terms of the inference engine running time, is shown in Figure 11. For each $(benchmark, analysis, \#feedback)$ configuration, the running time shown is an average over the three runs of the corresponding configuration. We observe two major trends from this figure. First, as expected, the running time is dependent on the size of the benchmark and the complexity of the analysis. For both the analyses in the control study, Eugene takes the longest time for `avrora`, our largest benchmark. Also, for each of our benchmarks, the *datarace* analysis, with fewer rules, needs shorter time. Recollect that Eugene uses an off-the-shelf solver for solving the constraints of probabilistic analysis, and thus the performance of the inference engine largely depends on the performance of the underlying solver. The running time of all such solvers depends on the number of ground clauses that are generated, and this number in turn depends on the size of the input program and complexity of the analysis.

Second, the amount of feedback does not significant affect running time. Incorporating the feedback only requires adding the liked/disliked queries as soft rules, and thus does not significantly alter the underlying set of constraints.

Finally, the fact that Eugene spends up to 120 minutes (*polysite* analysis on `avrora` with 15% feedback) might seem disconcerting. But note that this represents the time spent by the *system* rather than the *user*, in computing the new results after incorporating the user feedback. Since Eugene uses the underlying solver as a black-box, any improvement in solver technology directly translates into improved performance of Eugene. Given the variety of solvers that already exist [8, 14, 26, 33, 34, 37], and the ongoing active research in this area, we expect the running times to improve further.

Answer to **RQ2**: Eugene effectively scales to large programs up to a few hundred KLOC, and its scalability will only improve with advances in underlying solver technology. Additionally, the amount of feedback has no significant effect on the scalability of Eugene.

## 5.4 Usability of Eugene

In this section, we evaluate the results of our user study conducted using Eugene. The usage model for Eugene assumes that analysis users are familiar with the kind of reports produced by the analysis as well as with the program under analysis. To ensure familiarity with reports produced by *infoflow* analysis, we informed all our users about the expected outcomes of a precise *infoflow* analysis in general. However, familiarity with the program under analysis is harder to achieve and typically requires the user to have spent time developing or fixing the program. To address this issue, we choose relatively smaller benchmarks in our study that users can understand without too much effort or expertise. The users in this study were not informed about the internal working of either Eugene or the *infoflow* analysis.

The two main questions that we evaluate here are: (i) the ease with which users are able to analyze the reported results and provide feedback, and (ii) the quality of the user feedback. To answer the first question, we record the time spent by each user in analyzing the *infoflow* reports and providing the feedback for each benchmark. Recall that we ask each user to provide eight reports as feedback, labeled either spurious or real. Figure 12 shows the time spent by each user on analyzing the reports and providing feedback. We observe that the average time spent by the users is only 8 minutes on `secbench1`, 11.5 minutes on `secbench2`, and 5.5 minutes on `secbench3`. These numbers show that the users are able to inspect the analysis output and provide feedback to Eugene with relative ease on these benchmarks.

To evaluate the quality of the user provided feedback, we consider the precision of Eugene when it is run on the probabilistic version of *infoflow* analysis with the feedback. Figure 13 shows the false bugs eliminated and the true bugs retained by Eugene for each user and benchmark. This figure is similar in format to Figures 8 and 9. However, for each benchmark, instead of different bars representing a different amount of feedback, the different bars here represent different users, with feedback amount fixed at eight reports. The varying behavior of Eugene on these benchmarks highlights the strengths and limits of our approach.

For `secbench1`, an average of 78% of the false reports are eliminated and 62.5% of the true reports are retained. The important thing to note here is that the number of true reports retained is sensitive to the user feedback. With the right feedback, all the true reports are retained ($5^{th}$ bar). However, in the case where the user only chooses to provide one true feedback report ($4^{th}$ bar), Eugene fails to retain most of the true reports.

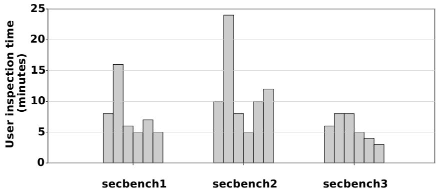


**Figure 12: Time spent by each user in inspecting reports of *infoflow* analysis and providing feedback.**

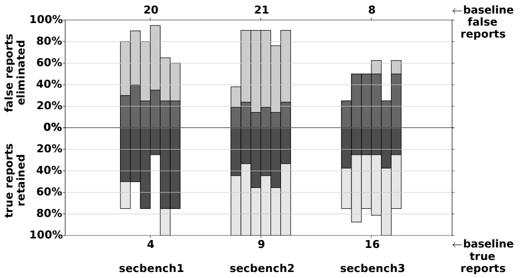


**Figure 13: Results of Eugene on *infoflow* analysis with real user feedback. Each bar maps to a user.**

For `secbench2`, an average of 79% of the false reports are eliminated and 100% of the true reports are retained. The reason Eugene does well here is that `secbench2` has multiple large clusters of reports with the same root cause. User feedback on any report in such clusters generalizes to other reports in the cluster. This highlights the fact that Eugene tends to produce more precise results when there are larger clusters of reports with the same root cause.

For `secbench3`, an average of 46% of the false reports are eliminated while 82% of the true reports are retained. First, notice that this benchmark produces only eight false reports. We traced the relatively poor performance of Eugene in generalizing the feedback on false reports to limiting the analysis user's interaction with the system to liking or disliking the results. This does not suffice for `secbench3` because, to effectively suppress the false reports in this case, the user must add new analysis rules. We intend to explore this richer interaction model in future work.

Finally, we observed that for all the benchmarks in this study, the labels provided by the users to the feedback reports matched with the ground truth. While this is not unexpected, it is important to note that Eugene is robust even under incorrectly labeled feedback, and can produce precise answers if a majority of the feedback is correctly labeled.

Answer to **RQ3**: It is feasible for users to inspect analysis output and provide feedback to Eugene since they only needed an average of 8 minutes for this activity in our user study. Further, in general, Eugene produce precise results with this user feedback, leading to the conclusion that it is not unreasonable to expect useful feedback from users.

### 5.5 Limitations of Eugene

Eugene requires analyses to be specified using the Datalog-based language described in § 3. Additionally, the program to be analyzed itself has to be encoded as a set of ground facts. This choice is motivated by the fact that a growing number of program analysis tools including bddbddb [43], Chord [29], Doop [39], LLVM [36], Soot [21], and Z3 [12] support specifying analyses and programs in Datalog.

The offline (learning) component of Eugene requires the analysis designer to specify which analysis rules must be soft. Existing analyses employ various approximations such as path-, flow-, and context-insensitivity; in our experience, rules encoding such approximations are good candidates for soft rules. Further, the learning component requires suitable training data in the form of desired analysis output. We expect such training data to be either annotated by the user, or generated by running a precise but unscalable version of the same analysis on small sample programs. Learning using partial or noisy training data is an interesting future direction that we plan to explore.

## 6. RELATED WORK

Our work is related to existing work on classifying error reports and other applications of probabilistic reasoning.

Dillig et al. [9] propose a user-guided approach to classify reports of analyses as errors or non-errors. They use abductive inference to compute small, relevant queries to pose to a user that capture exactly the information needed to discharge or validate an error. Their approach does not incorporate user feedback into the analysis specification and generalize it to other reports. Blackshear and Lahiri [6] propose a post-processing framework to prioritize alarms produced by a static verifier based on semantic reasoning of the program. Statistical error ranking techniques [13, 15, 16] employ statistical methods and heuristics to rank errors reported by an underlying static analysis. Non-statistical clustering techniques correlate error reports based on a root-cause analysis [18, 19]. Our technique, on the other hand, makes the underlying analysis itself probabilistic.

Recent years have seen many of applications of probabilistic reasoning to analysis problems. In particular, specification inference techniques based on probabilistic inference [4, 17, 24] can be formulated as probabilistic analyses (as defined in § 3). It would be interesting to explore the possibility of solving these specification inference formulations using the algorithms proposed in this paper. Another connection between user-guided program analysis and specification inference is that user feedback can be looked upon as an iterative method by means of which the analysis user communicates a specification to the program analysis tool. Finally, the inferred specifications can themselves be employed as soft rules in our system.

## 7. CONCLUSION

We presented a user-guided approach to program analysis that shifts decisions about the kind and degree of approximations to apply in analyses from analysis writers to analysis users. Our approach enables users to interact with the analysis by providing feedback on a portion of the results produced by the analysis, and automatically uses the feedback to guide the analysis approximations to the user's preferences. We implemented our approach in a system Eugene and evaluated it on real users, analyses, and programs. We showed that Eugene greatly reduces misclassified reports even with limited amounts of user feedback.

**Acknowledgements.** We thank the referees for useful feedback. This work was supported by DARPA under agreements #FA8750-12-2-0020 and #FA8750-15-2-0009, and by NSF award #1253867. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.

# 8. REFERENCES

[1] Apache FTP Server. `http://mina.apache.org/ftpserver-project/`.

[2] PJBench. `https://code.google.com/p/pjbench/`.

[3] Securibench Micro. `http://suif.stanford.edu/~livshits/work/securibench-micro/index.html`.

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

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[6] S. Blackshear and S. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *PLDI*, 2013.

[7] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.

[8] A. Chaganty, A. Lal, A. Nori, and S. Rajamani. Combining relational learning with SMT solvers using CEGAR. In *CAV*, 2013.

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

[10] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.

[11] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.

[12] K. Hoder, N. Bjørner, and L. M. de Moura. *$\mu Z$*- an efficient engine for fixed points with constraints. In *CAV*, 2011.

[13] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *SAS*, 2005.

[14] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The alchemy system for statistical relational AI. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2007.

[15] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *FSE*, 2004.

[16] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, 2003.

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

[18] W. Le and M. L. Soffa. Path-based fault correlations. In *FSE*, 2010.

[19] W. Lee, W. Lee, and K. Yi. Sound non-statistical clustering of static analysis alarms. In *VMCAI*, 2012.

[20] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, 2002.

[21] O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI*, 2004.

[22] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *CC*, 2006.

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

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

[25] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *APLAS*, 2005.

[26] R. Mangal, X. Zhang, M. Naik, and A. Nori. Solving weighted constraints with applications to program analysis. `http://hdl.handle.net/1853/53191`, 2015.

[27] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[28] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, 14(1), 2005.

[29] M. Naik. Chord: A program analysis platform for Java. `http://jchord.googlecode.com/`.

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

[31] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.

[32] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[33] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. In *VLDB*, 2011.

[34] J. Noessner, M. Niepert, and H. Stuckenschmidt. RockIt: Exploiting parallelism and symmetry for MAP inference in statistical relational models. In *AAAI*, 2013.

[35] C. H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[36] E. I. Psallida. Relational representation of the LLVM intermediate language. B.S. Thesis, University of Athens, Jan. 2014.

[37] S. Riedel. Improving the accuracy and efficiency of MAP inference for Markov Logic. In *UAI*, 2008.

[38] P. Singla and P. Domingos. Discriminative training of markov logic networks. In *AAAI*, 2005.

[39] Y. Smaragdakis and M. Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog 2.0 Workshop*, 2010.

[40] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2013.

[41] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: context-sensitivity, across the board. In *PLDI*, 2014.

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

[43] J. Whaley, D. Avots, M. Carbin, and M. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, 2005.

[44] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

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