User-Guided Program Reasoning using Bayesian Inference

Mukund Raghothaman
University of Pennsylvania, USA
rmukund@cis.upenn.edu

Sulekha Kulkarni
University of Pennsylvania, USA
sulekha@cis.upenn.edu

Kihong Heo
University of Pennsylvania, USA
kheo@cis.upenn.edu

Mayur Naik
University of Pennsylvania, USA
mhnaik@cis.upenn.edu

Abstract
Program analyses necessarily make approximations that often lead them to report true alarms interspersed with many false alarms. We propose a new approach to leverage user feedback to guide program analyses towards true alarms and away from false alarms. Our approach associates each alarm with a confidence value by performing Bayesian inference on a probabilistic model derived from the analysis rules. In each iteration, the user inspects the alarm with the highest confidence and labels its ground truth, and the approach recomputes the confidences of the remaining alarms given this feedback. It thereby maximizes the return on the effort by the user in inspecting each alarm. We have implemented our approach in a tool named Bingo for program analyses expressed in Datalog. Experiments with real users and two sophisticated analyses—a static datarace analysis for Java programs and a static taint analysis for Android apps—show significant improvements on a range of metrics, including false alarm rates and number of bugs found.

CCS Concepts • Software and its engineering → Automated static analysis; • Mathematics of computing → Bayesian networks; • Information systems → Retrieval models and ranking;

Keywords Static analysis, belief networks, Bayesian inference, alarm ranking

ACM Reference Format:

1 Introduction
Diverse forms of program reasoning, including program logics, static analyses, and type systems, all rely on logical modes of deriving facts about programs. However, due to classical reasons such as undecidability and practical considerations such as scalability, program reasoning tools are limited in their ability to accurately deduce properties of the analyzed program. When the user finally examines the tool output to identify true alarms, i.e. properties which the program actually fails to satisfy, her experience is impaired by the large number of false alarms, i.e. properties which the tool is simply unable to prove.

It is well-known that alarms produced by program reasoning tools are correlated: multiple true alarms often share root causes, and multiple false alarms are often caused by the tool being unable to prove some shared intermediate fact about the analyzed program. This raises the possibility of leveraging user feedback to suppress false alarms and increase the fraction of true alarms presented to the user. Indeed, a large body of previous research is aimed at alarm clustering [35, 36], ranking [30, 31], and classification [25, 41, 61].

In this paper, we fundamentally extend program analyses comprising logical rules with probabilistic modes of reasoning. We do this by quantifying the incompleteness of each deduction rule with a probability, which represents our belief that the rule produces invalid conclusions despite having valid hypotheses. Instead of just a set of alarm reports, we now additionally obtain confidence scores which measure our belief that the alarm represents a real bug. Furthermore, we are able to consistently update beliefs in response to new information obtained by user feedback. By intelligently selecting reports to present to the user for inspection, and by incorporating user feedback in future iterations, we have the
potential to greatly improve the practical utility of program reasoning tools.

We concretize these ideas in the context of analyses expressed in Datalog, a logic programming language which is increasingly used to declaratively specify complex program analyses [2, 3, 7, 40, 56, 60]. Inspired by the literature on probabilistic databases [11, 18], we develop a technique to convert Datalog derivation graphs into Bayesian networks. Computing alarm confidences can then be seen as performing marginal inference to determine the posterior probabilities of individual alarms conditioned on user feedback. By embedding the belief computation directly into the derivation graph, our technique exploits correlations between alarms at a much finer granularity than previous approaches.

We have implemented our technique in a tool named Bingo and evaluate it using two state-of-the-art analyses: a static datarace analysis [49] on a suite of 8 Java programs each comprising 95–616 KLOC, and a static taint analysis [15] on a suite of 8 Android apps each comprising 40–98 KLOC. We compare Bingo to two baselines: a random ranker, BASE-R, in which the user inspects each alarm with uniform probability, and a ranker based on a sophisticated alarm classification system [41], Base-C. On average, to discover all true alarms, the user needs to inspect 58.5% fewer alarms with Bingo compared to BASE-R, and 44.2% fewer alarms compared to BASE-C. We also conduct a user study with 21 Java programmers and confirm that small amounts of incorrect user feedback do not significantly affect the quality of the ranking, or overly suppress the remaining true alarms.

In summary, this paper makes the following contributions:

- A systematic methodology to view Datalog derivation graphs as Bayesian networks and thereby attach confidence scores, i.e., beliefs, to individual alarms, and update these values based on new user feedback.
- A framework to rank program analysis alarms based on user feedback. While we principally target analyses expressed in Datalog, the approach is extendable to any analysis algorithm written in a deductive style.
- Theoretical analysis showing that, assuming a probabilistic model of alarm creation, marginal probabilities provide the best way to rank alarms.
- Empirical evaluation with realistic analyses on large programs and a user study showing that Bingo significantly outperforms existing approaches on a range of metrics.

2 Motivating Example

Consider the Java program in Figure 1, adapted from the open-source Apache FTP Server. The program creates a new RequestHandler thread for each connection, and concurrently runs a TimerThread in the background to clean up idle connections. Multiple threads may simultaneously call the getRequest() and close() methods (from lines 28 and 48), and different threads may also simultaneously call close() (from lines 28 and 48).

Dataraces are a common and insidious kind of error that plague multi-threaded programs. Since getRequest() and close() may be called on the same RequestHandler object by different threads in parallel, there exists a datarace between the lines labelled L4 and L7: the first thread may read the request field while the second thread concurrently sets the request field to null.

On the other hand, even though the close() method may also be simultaneously invoked by multiple threads on the same RequestHandler object, the atomic test-and-set operation on lines L1–L3 ensures that for each object instance, lines L4–L7 are executed at most once. There is therefore no datarace between the pair of accesses to controlSocket on lines L4 and L5, and similarly no datarace between the accesses to request (lines L6 and L7).

We may use a static analysis to find dataraces in this program. However, due to the undecidable nature of the problem, the analysis may also report alarms on lines L4–L7. In the rest of this section, we illustrate how Bingo generalizes from user feedback to guide the analysis away from the false positives and towards the actual datarace.

2.1 A Static Datarace Analysis

Figure 2 shows a simplified version of the analysis in Chord, a static datarace detector for Java programs [49]. The analysis is expressed in Datalog as a set of logical rules over relations.

The analysis takes the relations N(p1, p2), U(p1, p2), and A(p1, p2) as input, and produces the relations R(p1, p2) and race(p1, p2) as output. In all relations, variables p1 and p2 range over the domain of program points. Each relation may be visualized as the set of tuples indicating some known facts about the program. For example, for the program in Figure 1, N(p1, p2) may contain the tuples N(L1, L2), N(L2, L3), etc. While some input relations, such as N(p1, p2), may be directly obtained from the text of the program being analyzed, other input relations, such as U(p1, p2) or A(p1, p2), may themselves be the result of earlier analyses (in this case, a lockset analysis and a pointer analysis, respectively).

The rules are intended to be read from right-to-left, with all variables universally quantified, and the − operator interpreted as implication. For example, the rule r1 may be read as saying, "For all program points p1, p2, p3, if p1 and p2 may execute in parallel (P(p1, p2)), and p3 may be executed immediately after p2 (N(p2, p3)), and p1 and p3 are not guarded by a common lock (U(p1, p3)), then p1 and p3 may themselves execute in parallel."

Observe that the analysis is flow-sensitive, i.e., it takes into account the order of program statements, represented by the relation N(p1, p2), but path-insensitive, i.e., it disregards the satisfiability of path conditions and predicates along branches. This is an example of an approximation to enable the analysis to scale to large programs.
2.2 Applying the Analysis to a Program

To apply the analysis of Figure 2 to the program in Figure 1, one starts with the set of input tuples, and repeatedly applies the inference rules \( r_1 \), \( r_2 \), and \( r_3 \), until no new facts can be derived. Starting with the tuple \( P(L4, L2) \), we show a portion of the derivation graph thus obtained in Figure 3. Each box represents a tuple, and is shaded gray if it is an input tuple. Nodes identified with rule names represent grounded clauses: for example, the node \( r_1(L4, L2, L3) \) indicates the "grounded instance" of the rule \( r_1 \) with \( p_1 = L4, p_2 = L2, \) and \( p_3 = L3 \). This clause takes as hypotheses the tuples \( P(L4, L2), N(L2, L3), \) and \( U(L4, L3), \) and derives the conclusion \( P(L4, L3) \), and the arrows represent these dependencies.

Observe that clause nodes are conjunctive: a rule fires iff all of its antecedents are derivable. On the other hand, consider another portion of the derivation graph in Figure 4. The tuple \( P(L6, L7) \) can be derived in one of two ways: either by instantiating \( r_1 \) with \( p_1 = L6, p_2 = L6, \) and \( p_3 = L7 \), or by instantiating \( r_2 \) with \( p_1 = L7 \) and \( p_2 = L6 \). Tuple nodes are therefore disjunctive: a tuple is derivable iff there exists at least one derivable clause of which it is the conclusion.

Observe that lines \( L4 \) and \( L2 \) can indeed execute in parallel, and the original conclusion \( P(L4, L2), \) in Figure 3, is true. However, the subsequent conclusion \( P(L4, L3) \) is spurious, and is caused by the analysis being incomplete: the second thread to enter the \( synchronized \) block will necessarily leave the method at line \( L2 \). Four subsequent false alarms—\( race(L4, L5), race(L5, L5), race(L5, L7), \) and \( race(L7, L7) \)—all result from the analysis incorrectly concluding \( P(L4, L3) \).
We also continue to treat tuple nodes as regular disjunctions: 

\[ Pr(P(L, L, L) \mid r_1(L, L, L)) = 1, \quad \text{and} \quad Pr(P(L, L, L) \mid \neg r_1(L, L, L)) = 0. \]

We also continue to treat tuple nodes as regular disjunctions: 

\[ Pr(P(L, L, L) \mid r_1(L, L, L)) = 1, \quad \text{and} \quad Pr(P(L, L, L) \mid \neg r_1(L, L, L)) = 0. \]

and treat all input tuples \( t \) as being known with certainty: 
\[ Pr(t) = 1. \]

We discuss how to learn these rule probabilities in Section 3.3. For now, we associate the rule \( r_2 \) with firing probability \( p_2 = 0.95 \), and \( r_2 \) with probability \( p_2 = 1.00 \). Finally, to simplify the discussion, we treat \( Pr(P(L, L, L)) = 0.40 \) and \( Pr(P(L, L, L)) = 0.60 \).

### 2.4 From Derivation Graphs to Bayesian Networks

By attaching conditional probability distributions (CPDs) such as equations 1–6 to each node of Figure 3, Bingo views the derivation graph as a Bayesian network. Specifically, Bingo performs marginal inference on the network to associate each alarm with the probability, or belief, that it is a true datarace. This procedure generates a list of alarms ranked by probability, shown in Table 1a. For example, it computes the probability of race\((L, L, L)\) as follows:

\[ Pr(\text{race}(L, L, L)) = Pr(\text{race}(L, L, L) \land r_3(L, L, L)) \]
\[ + Pr(\text{race}(L, L, L) \land \neg r_3(L, L, L)) \]
\[ = Pr(\text{race}(L, L, L) \land r_3(L, L, L)) \]
\[ = Pr(\text{race}(L, L, L) \mid r_3(L, L, L)) \cdot Pr(r_3(L, L, L)) \]
\[ = Pr(\text{race}(L, L, L) \mid P(L, L, L) \land A(L, L, L)) \]
\[ \cdot Pr(P(L, L, L)) \cdot Pr(A(L, L, L)) \]
\[ = 0.95 \cdot Pr(P(L, L, L)) = 0.95^4 \cdot Pr(P(L, L, L)) \]
\[ = 0.95^8 \cdot Pr(P(L, L, L)) = 0.398. \]

The user now inspects the top-ranked report, race\((L, L, L)\), and classifies it as a false alarm. The key idea underlying Bingo is that generalizing from feedback is conditioning on evidence. By replacing the prior belief \( Pr(a) \), for each alarm \( a \), with the posterior belief, \( Pr(a \mid \neg \text{race}(L, L, L)) \), Bingo effectively propagates the user feedback to the remaining conclusions of the analysis. This results in the updated list of alarms shown in Table 1b. Observe that the belief in the closely related alarm race\((L, L, L)\) drops from 0.324 to 0.030, while the belief in the unrelated alarm race\((L, L, L)\) remains unchanged at 0.279. As a result, the entire family of false alarms drops in the ranking, so that the only true datarace is now at the top.

The computation of the updated confidence values occurs by a similar procedure as before. For example:

\[ Pr(\text{race}(L, L, L) \mid \neg \text{race}(L, L, L)) \]
Finally, by Bayes’ rule, we have:
\[ \Pr(P(L, L, L) | \neg \text{race}(L, L, L)) = 0.95^3 \cdot \Pr(P(L, L, L) | \neg \text{race}(L, L, L)) \cdot \Pr(P(L, L, L)) \]
\[ = \frac{0.05 \cdot 0.95^2 \cdot 0.60}{0.60} = 0.03. \]

Our prior belief in \( P(L, L, L) \) was \( \Pr(P(L, L, L)) = 0.42 \), so that \( \Pr(P(L, L, L) | \neg \text{race}(L, L, L)) \ll \Pr(P(L, L, L)) \), but is still strictly greater than 0. This is because one eventuality by which \( \neg \text{race}(L, L, L) \) may occur is for \( P(L, L, L) \) to be true, but for the clause \( r_3(L, L, L) \) to misfire. We may now conclude that \( \Pr(\text{race}(L, L, L) | \neg \text{race}(L, L, L)) = 0.95^3 \cdot 0.03 = 0.030. \)

### 2.5 The Interaction Model

In summary, given an analysis and a program to be analyzed, Bingo takes as input the set of tuples and grounded clauses produced by the Datalog solver at fixpoint, and constructs the belief network. Next, it performs Bayesian inference to compute the probability of each alarm, and presents the alarm with highest probability for inspection by the user. The user then indicates its ground truth, and Bingo incorporates this feedback as evidence for subsequent iterations. We summarize this process in Figure 5.

There are several possible stopping criteria by which the user could cease interaction. She could choose to only inspect alarms with confidence higher than some threshold \( p_0 \), and stop once the confidence of the highest ranked alarm drops below \( p_0 \). Alternatively, she could choose to only inspect \( n \) alarms, and stop after \( n \) iterations. Of course, in all these situations, we would lose any soundness guarantees provided by the underlying analysis, but given the large number of alarms typically emitted by analysis tools and the time constraints frequently placed on developers, and as evidenced by our experiments in Section 5, Bingo promises to form a valuable component of the quality control toolchain.

### 3 Framework for Bayesian Inference

We formally describe the workflow of the previous section in Algorithm 1. We use an off-the-shelf solver [45] for the conditional probability queries in step 7(b). In this section and the next, we discuss the main technical ideas in our paper, spanning lines 3–5.

#### Algorithm 1 Bingo\((D, P, p)\), where \( D \) is the analysis expressed in Datalog, \( P \) is the program to be analyzed, and \( p \) maps each analysis rule \( r \) to its firing probability \( p_r \).

1. Let \( I = \text{INPUTRELATIONS}_D(P) \). Populate all input relations \( I \) using the program text and prior analysis results.
2. Let \((C, A, GC) = \text{DATALOSOLVE}(D, I)\). \( C \) is the set of output tuples, \( A \subseteq C \) is the set of alarms produced, and \( GC \) is the set of grounded clauses.
3. Compute \( GC_\alpha := \text{CYCLEELIM}(I, GC) \). Eliminate cycles from the grounded constraints.
4. (Optionally) Update \( GC_\alpha := \text{OPTIMIZE}(I, GC_\alpha, A) \). Reduce the size of the set of grounded constraints.
5. Construct Bayesian network \( BN \) from \( GC_\alpha \) and \( p \), and let \( \Pr \) be its joint probability distribution.
6. Initialize the feedback set \( e := \emptyset \).
7. While there exists an unlabelled alarm:
   a. Let \( A_u = A \setminus e \) be the set of all unlabelled alarms.
   b. Determine the top-ranked unlabelled alarm:
      \[ a_t = \arg \max_{a \in A_u} \Pr(a | e). \]
   c. Present \( a_t \) for inspection. If the user labels it a true alarm, update \( e := e \cup \{a_t\} \). Otherwise, update \( e := e \cup \{\neg a_t\} \).

#### 3.1 Preliminaries

We begin this section by briefly recapping the semantics of Datalog and Bayesian networks. For a more detailed treatment, we refer the reader to [1] and [29].

**Datalog.** We fix a collection \( \{R, S, T, \ldots\} \) of relations. Each relation \( R \) has an arity \( k \), and is a set of tuples \( R(v_1, v_2, \ldots, v_k) \), where the atoms \( v_1, v_2, \ldots, v_k \) are drawn from appropriate domains. Examples include the relations \( P \) ("may happen in parallel"), \( N \) ("may execute after"), and \( A \) ("may alias") from Figure 2. The analyst divides these into input and output relations, and specifies the computation of the output relations using a set of rules, each of which is of the form:

\[ R_h(u_h) \leftarrow R_1(u_1), R_2(u_2), \ldots, R_p(u_p), \]

where \( R_h \) is an output relation, and \( u_h, u_1, u_2, \ldots, u_p \) are free tuples. Examples include rules \( r_1, r_2, \) and \( r_3 \) in Figure 2. As mentioned before, each rule may be read as a universally quantified formula: "For all valuations \( u \) of the free variables, if \( R_1(u_1), R_2(u_2), \ldots, R_p(u_p) \), then \( R_h(u_h) \)."

Instantiating the free variables of a rule yields a Horn clause, \( R_1(u_1) \land R_2(u_2) \land \cdots \land R_p(u_p) \Rightarrow R_h(u_h) \). For example, the constraint \( r_1(L, L, L, L, L) \) from Figure 3 represents the Horn clause \( P(L, L, L) \land N(L, L, L) \land U(L, L, L) \Rightarrow P(L, L, L) \).

To solve a Datalog program, we accumulate the grounded constraints until fixpoint. Given a valuation \( I \) of all input relations, we initialize the set of conclusions, \( C \simeq I \), and initialize the grounded constraints to the set of input tuples, \( GC := \{ \text{true} \Rightarrow t \mid t \in I \} \). We repeatedly apply each rule to update \( C \) and \( GC \) until no new conclusions can be
Table 1. List of alarms produced by Bingo, (a) before, and (b) after the feedback \( \neg \text{race}(L4, L5) \). Observe how the real datarace \( \text{race}(L0, L7) \) rises in the ranking as a result of feedback.

(a) \( \Pr(a) \).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Belief</th>
<th>Program points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.398</td>
<td>RequestHandler : L4, RequestHandler : L5</td>
</tr>
<tr>
<td>2</td>
<td>0.378</td>
<td>RequestHandler : L5, RequestHandler : L5</td>
</tr>
<tr>
<td>3</td>
<td>0.324</td>
<td>RequestHandler : L6, RequestHandler : L7</td>
</tr>
<tr>
<td>4</td>
<td>0.308</td>
<td>RequestHandler : L7, RequestHandler : L7</td>
</tr>
<tr>
<td>5</td>
<td>0.279</td>
<td>RequestHandler : L0, RequestHandler : L7</td>
</tr>
</tbody>
</table>

(b) \( \Pr(a \mid \neg \text{race}(L4, L5)) \).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Belief</th>
<th>Program points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.279</td>
<td>RequestHandler : L0, RequestHandler : L7</td>
</tr>
<tr>
<td>2</td>
<td>0.035</td>
<td>RequestHandler : L5, RequestHandler : L5</td>
</tr>
<tr>
<td>3</td>
<td>0.030</td>
<td>RequestHandler : L6, RequestHandler : L7</td>
</tr>
<tr>
<td>4</td>
<td>0.028</td>
<td>RequestHandler : L7, RequestHandler : L7</td>
</tr>
<tr>
<td>5</td>
<td>0.0</td>
<td>RequestHandler : L4, RequestHandler : L5</td>
</tr>
</tbody>
</table>

Bayesian networks. We will only consider boolean-valued random variables, and specialize our definition for this purpose. Fix a set of random variables \( V \), and a directed acyclic graph \( G = (V, E) \) with the random variables \( V \) as its vertices. Given \( v \in V \), we write \( \text{Pa}(v) \) for the set of variables with edges leading to \( v \). Formally,

\[
\text{Pa}(v) = \{ u \in V \mid u \rightarrow v \in E \}.
\]

The conditional probability distribution (CPD) of a random variable \( v \) is a function which maps concrete valuations \( x_{\text{Pa}(v)} \) of \( \text{Pa}(v) \) to the conditional probability of the event \( v = \text{True} \), and we write this as \( p(v \mid x_{\text{Pa}(v)}) \). Naturally, the complementary event \( v = \text{False} \) has conditional probability \( p(\neg v \mid x_{\text{Pa}(v)}) = 1 - p(v \mid x_{\text{Pa}(v)}) \). The Bayesian network, given by the triple \( BN = (V, G, p) \), is essentially a compact representation of the following joint probability distribution:

\[
\Pr(x) = \prod_v p(x_v \mid x_{\text{Pa}(v)}),
\]

where the joint assignment \( x \) is a valuation \( x_v \) for each \( v \in V \), and \( x_{\text{Pa}(v)} \) is the valuation restricted to the parents of \( v \). From Equation 7, we may readily define other quantities of interest, including the marginal probability of a variable, \( \Pr(v) = \sum_{x \mid x_v = \text{True}} \Pr(x) \), \( \Pr(\neg v) = 1 - \Pr(v) \), and the conditional probability of arbitrary events: \( \Pr(v \mid e) = \Pr(v \land e) / \Pr(e) \).

3.2 From Derivation Graphs to Bayesian Networks

The set of conclusions \( C \) and the grounded clauses \( GC \) from a Datalog program at fixpoint naturally induce a graph \( G(C, GC) \) over the vertices \( C \cup GC \), with an edge from a tuple \( t \in C \) to a clause \( g \in GC \) whenever \( t \) is an antecedent of \( g \), and an edge from \( g \) to \( t \) whenever \( t \) is the conclusion of \( g \). We have already seen portions of this graph in Figures 3 and 4. Our goal is to view this graph as a Bayesian network, with each vertex \( v \in C \cup GC \) as a random variable, and by attaching CPDs such as those in Equations 1–6 to each node. Observe however, that while the underlying graph of the Bayesian network is required to be acyclic, the derivation graph may have cycles, such as that shown in Figure 6.

We therefore choose a subset \( GC_c \subseteq GC \) of clauses such that the induced graph is acyclic. We require that \( GC_c \) still derive every alarm \( a \in A \) originally produced by the analysis, and that the cycle elimination algorithm scale to the fixpoints produced by large programs (\( \approx 1–10 \) million grounded clauses). While we would like \( GC_c \) to be maximal, to capture as many correlations between alarms as possible, finding the largest \( GC_c \) which induces an acyclic graph \( G(C, GC_c) \) can be shown to be NP-complete by reduction from the maximum acyclic subgraph problem [19]. We therefore relax the maximality requirement for \( GC_c \), and use a modified version.
of the naive Datalog evaluator shown in Algorithm 2 as an efficient cycle elimination algorithm. The following theorem, proved in Appendix A, states that CycleElim(I, GC) satisfies the desired requirements:

**Theorem 3.1.** For all I and GC, if GC_c = CycleElim(I, GC), then (a) GC_c ⊆ GC, (b) every tuple derivable using GC is also derivable using GC_c, and (c) G(C, GC_c) is acyclic.

**Algorithm 2** CycleElim(I, GC), where I is the set of all input tuples, and GC is a set of grounded constraints.

1. Initialize the timestamp map T, such that for each tuple t, if t ∈ I, T(t) = 0, and otherwise, T(t) = ∞.
2. Given a grounded clause g, let A_g be the set of all its antecedents, and let c_g be its consequent.
3. While there exists a constraint g such that
   \[ T(c_g) > \max_{a \in A_g} T(a) + 1, \] update:
   \[ T(c_g) := \max_{a \in A_g} T(a) + 1. \]
4. Define GC_c = \{ g ∈ GC | T(c_g) > \max_{a ∈ A_g}(T(a)) \}. GC_c ⊆ GC is the set of all those constraints in GC whose consequent has a timestamp strictly greater than all of its antecedents. Return GC_c.

### 3.3 Learning Rule Firing Probabilities

Given a program P and an analysis D, the Bayesian network of the previous section is parameterized by the vector of rule probabilities, \( p \). To emphasize this dependence, we write \( Pr_p \) for the induced joint probability distribution.

We will now discuss how we may obtain \( p \). With a corpus of fully labelled data, the rule probability is simply the fraction of times the rule produces incorrect conclusions from true hypotheses. With a less extensively labelled dataset, one faces the challenge of latent (or unobserved) variables, which can be solved with the expectation maximization (EM) algorithm described below. However, both of these techniques require a large corpus, and a proper experimental evaluation involves partitioning the data into training and test sets. To avoid this problem in our experiments, we uniformly assign each rule to a probability of 0.999.

**The maximum likelihood estimator (MLE).** For a training program and its associated ground truth \( v \), our learning problem is to determine the “best” weight vector \( p \) which explains \( v \). One common measure of goodness is the likelihood, \( L(p; v) \), which may be informally motivated as the plausibility of \( p \) given the training data \( v \). Concretely, \( L(p; v) = Pr_p(v) \). To “learn” the rule firing probabilities is to then find the probability vector \( \hat{p} \) with highest likelihood:

\[ \hat{p} = \arg \max_p L(p; v). \]

**MLE by expectation maximization (EM).** In our setting, MLE may be performed by a straightforward implementation of the EM algorithm [29]. Starting with an arbitrary seed \( p^{(0)} \), the algorithm iteratively computes a sequence of estimates \( p^{(0)}, p^{(1)}, p^{(2)}, \ldots \), as follows:

\[ p_r^{(r+1)} = \frac{\sum_g Pr^{(r)}(c_g \land A_g) \cdot p_r^{(r)}(A_g)}{\sum_g p_r^{(r)}(A_g)}, \]

where \( g \) ranges over all grounded clauses associated with \( r \). The algorithm guarantees that \( L(p^{(r)}; v) \) monotonically increases with \( r \), which is also bounded above by 1, so that the procedure converges to a local maximum in practice.

### 3.4 Alarm Ranking in an Ideal Setting

We now consider some theoretical properties of the alarm ranking problem. We begin with the following question: How good is an ordering of alarms, \( w = a_1, a_2, \ldots, a_n \), in light of their associated ground truths, \( v_1, v_2, \ldots, v_n \)?

We use the number of inversions as a measure of ranking quality. A pair of alarms \( (a_i, a_j) \) from \( w \) forms an inversion if \( a_i \) appears before \( a_j \), but \( a_i \) is false and \( a_j \) is true, i.e. \( i < j \land \neg v_i \land v_j \). The ranker incurs a penalty for each inversion, because it has presented a false alarm before a real bug. Well ordered sequences of alarms usually have fewer inversions than poorly ordered sequences. We write \( \chi(w) \) for the number of inversions in \( w \).

Assume now that \( Pr(\cdot) \) describes the joint probability distribution of alarms. We seek the ordering of alarms with lowest expected inversion count. Two versions of this problem which are relevant in our setting: (a) alarm ranking given a fixed set of observations \( e \), and (b) alarm ranking in the interactive setting, where \( e \) grows with each iteration. The following theorem, which we prove in Appendix A, states that Bnngo-style ranking is optimal for the first problem.

**Theorem 3.2.** For each set of observations \( e \), the sequence \( w = a_1, a_2, \ldots, a_n \) of alarms, arranged according to decreasing \( Pr(a_i | e) \), has the minimum expected inversion count over all potential orderings \( w' \).

The second problem involves finding an optimal strategy in a Markov decision process [54] with \( O((n + 1)! \) states. Bnngo may be viewed as employing a greedy heuristic in this process, and we plan to investigate this significantly more challenging problem in future work.

### 4 Implementation

The central computational component of Bnngo is the conditional probability query \( Pr(a | e) \) in step 7(b) of Algorithm 1. We discharge these queries using the loopy belief propagation algorithm implemented in LibDAI [45]. Step 4 of Algorithm 1 is the result of combining the two optimizations we will now describe. We discuss additional engineering details involving the belief propagation algorithm in Appendix B.
Co-reachability based constraint pruning. While the solution to the Datalog program at fixpoint contains all derivable tuples, not all of these tuples are useful in the production of alarms. Therefore, our first optimization is to remove unnecessary tuples and clauses by performing a backward pass over GC. We initialize the set of useful tuples $U := A$, and repeatedly perform the following update until fixpoint:

$$U := U \cup \{ t \mid \exists g \in GC \text{ s.t. } t \in A_g \text{ and } c_g \in U \}.$$ 

Recall, from Algorithm 2, that we write $c_g$ for the conclusion of a grounded clause $g$, and $A_g$ for the set of all its antecedents. Informally, a tuple is useful if it is either itself an alarm, or can be used to produce a useful tuple. The pruned set of clauses may finally be defined as follows: $GC' = \{ g \in GC \mid c_g \in U \}$.

Chain compression. Consider the derivation graph shown in Figure 3, and observe the sequence of tuples $P(L4,L2) \rightarrow P(L4,L3) \rightarrow P(L4,L4) \rightarrow P(L4,L5)$. Both intermediate tuples $P(L4,L3)$ and $P(L4,L4)$ are produced by exactly one grounded clause, and are consumed as an antecedent by exactly one clause. Furthermore, since neither of them is an alarm node, we will never solicit feedback from the user about these tuples. We may therefore rewrite the derivation graph to directly conclude $P(L4,L2) \rightarrow P(L4,L5)$.

We formally present this optimization in Algorithm 3. The rewriting is essentially an eager application of the standard variable elimination procedure for Bayesian networks.

**Algorithm 3** COMPRESS($GC, C, A$), where $GC$ is the set of grounded constraints, $C$ is the set of conclusions, and $A$ is the set of alarms produced by the analysis.

1. For each tuple $t$, define:
   
   $$\text{SrCs}(t) := \{ g \in GC \mid t = c_g \},$$
   
   $$\text{SInKs}(t) := \{ g \in GC \mid t \in A_g \}.$$

2. Construct the following set:
   
   $$E := \{ t \in C \setminus A \mid |\text{SrCs}(t)| = 1 \land |\text{SInKs}(t)| = 1 \}.$$

3. While $E$ is not empty:
   a. Pick an arbitrary tuple $t \in E$, and let $\text{SrCs}(t) = \{ g_1 \}$, and $\text{SInKs}(t) = \{ g_2 \}$.
   b. Since $t = c_{g_1}$ and $t$ is an antecedent of $g_2$, let
      
      $$g_1 = a_1 \land a_2 \land \cdots \land a_k \implies t,$$
      
      $$g_2 = t \land b_1 \land b_2 \land \cdots \land b_p \implies t'. $$
   c. Define a new clause, $g' = a_1 \land a_2 \land \cdots \land a_k \land b_1 \land b_2 \land \cdots \land b_p \implies t'$. Update $GC := GC \cup \{ g' \} \setminus \{ g_1, g_2 \}$.
   d. If $g_1$ was associated with rule $r_1$ with probability $p_1$, and $g_2$ was associated with rule $r_2$ with probability $p_2$, then associate $g'$ with a new rule $r'$ with probability $p_1 p_2$.
4. Return $GC$.

5 Experimental Evaluation

Our evaluation seeks to answer the following questions:

- **Q1.** How effective is Bingo at ranking alarms?
- **Q2.** Can Bingo help in discovering new bugs missed by existing precise analysis tools?
- **Q3.** How robust is the ranking produced by Bingo and how adversely is it affected by incorrect responses?
- **Q4.** Does Bingo scale to large programs and how effective are the optimizations it employs?

We describe our experimental setup in Section 5.1, then discuss each of the above questions in Sections 5.2–5.5, and outline limitations of our approach in Section 5.6.

### 5.1 Experimental Setup

We conducted all our experiments on Linux machines with 3.0 GHz processors and 64 GB RAM running Oracle HotSpot JVM 1.6 and LibDAl version 0.3.2. We set a timeout of 2 hours per alarm proposed for both Bingo and the baselines.

#### 5.1.1 Instance Analyses

We summarize the key statistics of our instance analyses in Table 2, and describe them here in more detail.

**Table 2.** Statistics of the instance analyses.

<table>
<thead>
<tr>
<th></th>
<th>Analysis</th>
<th>Rules</th>
<th>Input relations</th>
<th>Output relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datarace</td>
<td>62</td>
<td>58</td>
<td>44</td>
<td>25</td>
</tr>
<tr>
<td>Taint</td>
<td>62</td>
<td>52</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

**Datarace analysis.** The datarace analysis [49] is built atop the Chord framework [48]. It combines thread-escape, may-happen-in-parallel, and lockset analyses that are flow-and-context sensitive. They build upon call-graph and aliasing information obtained from a 3-context-and-object sensitive but flow-insensitive pointer analysis [42]. The analysis is intended to be *sound* [39], i.e. sound with the exception of some difficult features of Java, including exceptions and reflection (which is resolved by a dynamic analysis [6]).

**Taint analysis.** The taint analysis [15] is built atop the Soot framework [59]. The variables of the analyzed program are associated with source and sink annotations, depending on whether it could contain data originating from a sensitive source or data that eventually flows to an untrusted sink, respectively. The analysis uses call-graph and aliasing information from a similar pointer analysis as before [42].

#### 5.1.2 Benchmarks

We evaluated Bingo on the suite of 16 benchmarks shown in Table 3. The first four datarace benchmarks are commonly used in previous work [13, 61], while we chose the remaining four from the DaCapo suite [4], and obtained the ground truth by manual inspection. The eight taint analysis benchmarks were chosen from the STAMP [15] repository, and are
User-Guided Program Reasoning using Bayesian Inference

a combination of apps provided by a major security company, and challenge problems used in past research.

5.1.3 Baseline Ranking Algorithms
We compare Bingo to two baseline algorithms, BASE-R and BASE-C. In each iteration, BASE-R chooses an alarm for inspection uniformly at random from the pool of unlabelled alarms. BASE-C is based on the alarm classifier Eugene [41]. We describe its operation in Appendix C.

5.1.4 User Study
To answer Q3, we conducted a user study to measure the fraction of alarms mislabelled by professional programmers. We placed an advertisement on upwork.com, an online portal for freelance programmers. We presented respondents with a tutorial on dataraces, and gave them a small 5-question test based on a program similar to that in Figure 1. Based on their performance in the test, we chose 21 of the 27 respondents, and assigned each of these developers to one of the benchmarks, hdec, ftp, webLech, and jsSpider. We gave them 20 alarms for labelling with an 8–10 hour time limit, such that each alarm was inspected by at least 5 independent programmers. To encourage thoughtful answers, we also asked them to provide simple explanations with their responses. We found that, for 90% of the questions, the majority vote among the responses resulted in the correct label.

5.2 Effectiveness of Ranking
One immediate way to measure the effectiveness of Bingo is to determine the rank at which the last true alarm is discovered. We present these statistics in Table 4. For example, the datarace analysis produces 522 alarms on ftp, of which 75 are real dataraces. Bingo presents all true alarms for inspection within just 103 rounds of interaction, compared to 368 for BASE-C, and 520 for BASE-R. Another notable example is 1uindex from the DaCapo suite, on which the analysis produces 940 alarms. Of these alarms, only 2 are real dataraces, and Bingo reports both bugs within just 14 rounds of interaction, compared to 101 for BASE-C and 587 for BASE-R. Over all benchmarks, on average, the user needs to inspect 44.2% and 58.5% fewer alarms than BASE-C and BASE-R respectively.

On the other hand, the last true alarm discovered may be an outlier and not representative of the entire interaction process. This is evident in the case of sunflow, for which one needs to inspect 838 of the 958 alarms produced to discover all bugs. Observe however, in the column RANK-90%-T, that the user discovers 90% of the true alarms within just 483 iterations. The more detailed comparison between Bingo and BASE-C presented in Figure 7 demonstrates that Bingo has a consistently higher yield of true alarms than BASE-C.

To capture this dynamical behavior, we consider the interaction process as represented by the ROC curve [14]. We plot this for the ftp benchmark in Figure 8. The x- and y-axes indicate the false and true alarms present in the benchmark, and each point (x, y) on the curve indicates an instant when the user has inspected x false alarms and y true alarms. At this step, if the system proposes a true alarm, then the next point on the curve is at (x, y + 1), and otherwise, the next point is at (x + 1, y). The solid line is the curve for Bingo, while the dotted lines are the ranking runs for each of the runs of BASE-C, and the diagonal line is the expected behavior of BASE-R. Observe that Bingo outperforms BASE-C not just in the aggregate, but across each of the individual runs.

Our final effectiveness metric, the AUC, is the normalized area under the ROC curve. The AUC is closely related to the inversion count $\chi$ of Section 3.4: if $n_t$ and $n_f$ are the number of true and false alarms in a sequence $w$ of alarms, then $\chi(w) = n_t n_f (1 - \text{AUC}(w))$. It can also be shown that the expected AUC of BASE-R is equal to 0.5. On a scale ranging from 0 to 1, on average, the AUC for Bingo exceeds that of BASE-C by 0.13 and of BASE-R by 0.37.

In summary, we conclude that Bingo is indeed effective at ranking alarms, and can significantly reduce the number of false alarms that the user needs to triage.

5.3 Discovering New Bugs
In an attempt to control the number of alarms produced, users are drawn to precise static and dynamic analysis tools, which promise low false positive rates [22]. However, as evidenced by missed security vulnerabilities such as Heartbleed [10], precise tools are necessarily unsound, and often miss important bugs. To check whether Bingo can help in this situation, we ran two state-of-the-art precise datarace detectors: Chord [49] with unsound flags turned on, and FastTrack [17], a dynamic datarace detector based on the happens-before relation. We ran FastTrack with the inputs that were supplied with the benchmarks.

We present the number of alarms produced and the number of bugs missed by each analyzer in Table 5. For example, by turning on the unsound options, we reduce the number of alarms produced by Chord from 522 to 211, but end up missing 39 real dataraces. Using Bingo, however, a user discovers all true alarms within just 103 iterations, thereby discovering 108% more dataraces while inspecting 51% fewer alarms.

Aggregating across all our benchmarks, there are 379 real dataraces, of which the unsound Chord analysis reports only 203 and produces 1,414 alarms. Bingo discovers all 379 dataraces within a total of just 1,736 iterations. The user therefore discovers 87% more dataraces by just inspecting 23% more alarms. In all, using Bingo allows the user to inspect 100 new bugs which were not reported either by FastTrack or by Chord in its unsound setting.

Furthermore, the analysis flags determine the number of alarms produced in an unpredictable way: reducing it from 958 alarms to 506 alarms for sunflow, but from 1,870 alarms to 80 alarms for xalan. In contrast, Bingo provides the user with much more control over how much effort they would like to spend to find bugs.
Table 3. Benchmark characteristics. ‘Total’ and ‘App’ columns are numbers using 0-CFA call graph construction, with and without the JDK for datarace analysis benchmarks, and with and without the Android framework for taint analysis benchmarks.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th># Classes</th>
<th># Methods</th>
<th>Bytecode (KLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdec</td>
<td>Web-crawler from ETH</td>
<td>357</td>
<td>44</td>
<td>2,154</td>
</tr>
<tr>
<td>ftp</td>
<td>Apache FTP server</td>
<td>499</td>
<td>119</td>
<td>2,754</td>
</tr>
<tr>
<td>weblech</td>
<td>Website download/mirror tool</td>
<td>579</td>
<td>56</td>
<td>3,344</td>
</tr>
<tr>
<td>jspider</td>
<td>Web spider engine</td>
<td>362</td>
<td>113</td>
<td>1,584</td>
</tr>
<tr>
<td>avror</td>
<td>AVR microcontroller simulator</td>
<td>52.3</td>
<td>11,097</td>
<td>3,875</td>
</tr>
<tr>
<td>luindex</td>
<td>Document indexing tool</td>
<td>1,168</td>
<td>169</td>
<td>7,494</td>
</tr>
<tr>
<td>sunflow</td>
<td>Photo-realistic image rendering system</td>
<td>1,857</td>
<td>127</td>
<td>12,934</td>
</tr>
<tr>
<td>xalan</td>
<td>XML to HTML transforming tool</td>
<td>1,727</td>
<td>390</td>
<td>12,214</td>
</tr>
</tbody>
</table>

Table 4. Summary of metrics for the effectiveness of Bingo. Rank-100%-T and Rank-90%-T are the ranks at which all and 90% of the true alarms have been inspected, respectively. For the baselines, we show the median measurement across five runs.

<table>
<thead>
<tr>
<th>Program</th>
<th>#Alarms</th>
<th>Rank-100%-T</th>
<th>Rank-90%-T</th>
<th>Area under the curve (AUC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdec</td>
<td>152</td>
<td>12 (7.89%)</td>
<td>67 (121)</td>
<td>143 (115) 135 (0.81) 0.76 0.50</td>
</tr>
<tr>
<td>ftp</td>
<td>522</td>
<td>75 (14.37%)</td>
<td>103 (368)</td>
<td>520 (80) 290 (0.98) 0.78 0.49</td>
</tr>
<tr>
<td>weblech</td>
<td>30</td>
<td>6 (20.00%)</td>
<td>11 (16)</td>
<td>29 (10) 15 (0.84) 0.78 0.48</td>
</tr>
<tr>
<td>jspider</td>
<td>257</td>
<td>9 (3.50%)</td>
<td>20 (128)</td>
<td>247 (19) 101 (0.97) 0.81 0.59</td>
</tr>
<tr>
<td>avror</td>
<td>978</td>
<td>29 (2.97%)</td>
<td>410 (971)</td>
<td>960 (365) 798 (0.75) 0.70 0.51</td>
</tr>
<tr>
<td>luindex</td>
<td>940</td>
<td>2 (0.21%)</td>
<td>14 (101)</td>
<td>587 (14) 101 (0.99) 0.89 0.61</td>
</tr>
<tr>
<td>sunflow</td>
<td>958</td>
<td>171 (17.85%)</td>
<td>838 (timeout)</td>
<td>952 (483) timeout 872 (0.79) timeout 0.50</td>
</tr>
<tr>
<td>xalan</td>
<td>1,870</td>
<td>75 (4.01%)</td>
<td>273 (timeout)</td>
<td>1844 (266) timeout 1,706 (0.91) timeout 0.50</td>
</tr>
</tbody>
</table>

5.4 Robustness of Ranking

One potential concern with tools such as Bingo is that mislabelled alarms will deteriorate their performance. Furthermore, concurrency bugs such as data races are notoriously hard to diagnose. However, in the user study described in Section 5.1, we observed that when a group of professional programmers are made to vote on the ground truth of an alarm, they are able to correctly classify 90% of the alarms.

We extrapolated the results of this study and simulated the runs of Bingo on ftp where the feedback labels had been corrupted with noise. In Table 6, we measure the ranks at which 90% and 100% of the alarms labelled true appear. As is expected of an outlier, the rank of the last true alarm degrades where the feedback labels had been inspected increases more gracefully, from 80 originally to 98 in the presence of 10% noise. In all cases, Bingo outperforms the original Base-C. We conclude that Bingo can robustly tolerate reasonable amounts of user error.

5.5 Scalability of the Ranking Algorithm

We present measurements of the running time of Bingo and of Base-C in Table 7. We also present the iteration time when
the optimizations of Section 4 are turned off. The iteration time, corresponding to one run of the belief propagation algorithm, is directly dependent on the size of the Bayesian network, which we indicate by the columns labelled #Tuples and #Clauses. In contrast, Base-C invokes a MaxSAT solver in each iteration, and the columns labelled #Vars and #Clauses indicate the size of the formula presented to the solver. Observe the massive gains in performance—on average, an improvement of 265×—as a result of the co-reachability based pruning and chain compression, because of which Bingo can handle even large benchmark programs such as xalan and sunflow, on which Base-C times out.

5.6 Limitations

The running time of Bingo could preclude its integration into an IDE. This limitation could be alleviated through solver improvements and modular reasoning techniques for large programs [34]. Secondly, we restrict ourselves to features already present in the analysis, which could potentially limit generalization from user feedback. Finally, early stopping criteria, such as those mentioned in Section 2.5, affect soundness and raise the possibility of missed bugs.

6 Related Work

There is a large body of research on techniques to overcome the incompleteness of automatic program reasoning tools. We catalog and summarize these efforts.

Interactive techniques. These approaches resolve alarms through user feedback; they reason about the logical structure of the analysis to identify queries that are then posed to the user [12, 35, 36, 51, 61]. Dillig et al [12] formulate the search for missing facts to discharge alarm as an abductive inference problem. Ivy [51] graphically displays succinct
counterexamples to the user to help identify inductive invariants. URSA [61] formulates the search for the root causes of false alarms in Datalog derivation graphs as an information gain maximization problem. Le and Soffa [35] and Lee et al [36] identify correlations between alarms of the form “If $a$ is false, then $b$ is also false.” Thus, an entire family of alarms can be resolved by the user inspecting a single alarm in the family. These techniques often give soundness guarantees assuming correct user responses. However, they are constrained by the limits of logical reasoning, and cannot extract fine correlations between alarms (such as, “If $a$ is false, then $b$ is also likely to be false.”). Moreover, in contrast to many of these systems, Bingo only asks the user to provide the ground truth of alarms they are already triaging, instead of posing other kinds of queries. This significantly lowers the burden of using our approach.

**Statistical techniques.** Such approaches leverage various kinds of program features to statistically determine which alarms are likely bugs. The z-ranking algorithm [30, 31] uses the observation that alarms within physical proximity of each other (e.g., within the same function or file) are correlated in their ground truth and applies a statistical technique called the z-test to rank alarms. More recently, other kinds of program features have been used to statistically classify analysis alarms [5, 25, 28, 58]. On the one hand, our work can be seen as explaining why these techniques tend to work; for instance, alarms within the same function are more likely to share portions of their derivation graph, and these correlations are therefore emergent phenomena that are naturally captured by our Bayesian network. On the other hand, these techniques can exploit extra-analytic program features, which is an important future direction for our work.

Previous techniques have also leveraged analysis-derived features, e.g., to assign confidence values to alarms [37], and classify alarms with a single round of user feedback [41]. The key difference compared to these works is Bingo’s ability to maximize the return on the user’s effort in inspecting each alarm. As described in Sections 3 and 5, Bingo therefore significantly outperforms the baseline based on [41].

Further out, there is a large body of work on using statistical techniques for mining likely specifications and reporting anomalies as bugs (e.g., [32, 38, 46, 55]) and for improving the performance of static analyzers (e.g., [8, 23, 24]).

**Inference techniques.** Starting with Pearl [52, 53], there is a rich body of work on using graphical models to combine logical and probabilistic reasoning in AI. Prominent examples include Bayesian networks and Markov networks, and we refer the reader to Koller and Friedman’s comprehensive textbook [29]. A more recent challenge involves extending these models to capture richer logical formalisms such as Horn clauses and first-order logic. This has resulted in frameworks such as probabilistic relational models [21], Markov logic networks [50, 57], Bayesian logic programs [26], and probabilistic languages such as Blog [43], ProbLog [16], and Infer.NET [44]. Our work may be viewed as an attempt to apply these ideas to program reasoning.

There are several methods to perform marginal inference in Bayesian networks. Examples include exact methods, such as variable elimination, the junction tree algorithm [27], and symbolic techniques [20]. approximate methods based on belief propagation [33, 45], and those based on sampling, such as Gibbs sampling or MCMC search. Recent advances on the random generation of SAT witnesses [9] also fall in this area. Given that exact inference is #P-complete [11, 29], our main consideration in choosing belief propagation was the desire for deterministic output, and the requirement to scale to large codebases.

### 7 Conclusion

In this paper, we presented Bingo, a static analysis tool which prioritizes true alarms over false ones by leveraging user feedback and by performing Bayesian inference over the derivation graph. We demonstrated significant improvement over two different baselines and across several metrics on two instance analyses and a suite of 16 benchmark programs.

We view Bingo as the first step in a program to integrate probabilistic techniques more deeply into traditional software engineering tools. Our immediate problems of interest include incrementality (How do we carry forward information from one version of the program being analyzed to the next?), and improving accuracy by considering program features which are invisible to the analysis (such as variable

---

**Table 7. Sizes of the Bayesian networks processed by Bingo, and of the MaxSAT problems processed by Base-C, and their effect on iteration time.** Data for the Android benchmarks is presented in Table 8 of Appendix C.

<table>
<thead>
<tr>
<th>Program</th>
<th>Bingo, optimized</th>
<th>Base-C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Tuples (K)</td>
<td>#Clauses (K)</td>
</tr>
<tr>
<td>hedd</td>
<td>2.2</td>
<td>3.1</td>
</tr>
<tr>
<td>ftp</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>weblech</td>
<td>0.31</td>
<td>0.38</td>
</tr>
<tr>
<td>jspider</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>avrora</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>luindex</td>
<td>5.3</td>
<td>6.5</td>
</tr>
<tr>
<td>sunflow</td>
<td>59</td>
<td>98</td>
</tr>
<tr>
<td>xalan</td>
<td>19</td>
<td>32</td>
</tr>
</tbody>
</table>

---

**Table 8. Summary of the Android benchmarks.**

<table>
<thead>
<tr>
<th>Program</th>
<th>#Vars (K)</th>
<th>#Clauses (K)</th>
<th>Iter time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hedd</td>
<td>3,007</td>
<td>50</td>
<td>35</td>
</tr>
<tr>
<td>ftp</td>
<td>35</td>
<td>9</td>
<td>45</td>
</tr>
<tr>
<td>weblech</td>
<td>37</td>
<td>29</td>
<td>59</td>
</tr>
<tr>
<td>jspider</td>
<td>240</td>
<td>26</td>
<td>497</td>
</tr>
<tr>
<td>avrora</td>
<td>2,067</td>
<td>559</td>
<td>2,182</td>
</tr>
<tr>
<td>luindex</td>
<td>3,636</td>
<td>373</td>
<td>46</td>
</tr>
<tr>
<td>sunflow</td>
<td>9,632</td>
<td>379</td>
<td>26,025</td>
</tr>
<tr>
<td>xalan</td>
<td>2,452</td>
<td>2,859</td>
<td>51,812</td>
</tr>
</tbody>
</table>

---

**Table 9. Summary of the full suite of benchmarks.**

<table>
<thead>
<tr>
<th>Program</th>
<th>#Vars (K)</th>
<th>#Clauses (K)</th>
<th>Iter time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hedd</td>
<td>3,007</td>
<td>50</td>
<td>35</td>
</tr>
<tr>
<td>ftp</td>
<td>35</td>
<td>9</td>
<td>45</td>
</tr>
<tr>
<td>weblech</td>
<td>37</td>
<td>29</td>
<td>59</td>
</tr>
<tr>
<td>jspider</td>
<td>240</td>
<td>26</td>
<td>497</td>
</tr>
<tr>
<td>avrora</td>
<td>2,067</td>
<td>559</td>
<td>2,182</td>
</tr>
<tr>
<td>luindex</td>
<td>3,636</td>
<td>373</td>
<td>46</td>
</tr>
<tr>
<td>sunflow</td>
<td>9,632</td>
<td>379</td>
<td>26,025</td>
</tr>
<tr>
<td>xalan</td>
<td>2,452</td>
<td>2,859</td>
<td>51,812</td>
</tr>
</tbody>
</table>
names and results of dynamic analyses). In the longer term, we plan to investigate various non-traditional uses of BINGO, such as during analysis design, to obtain data-driven insights into otherwise non-statistical analyses.

Acknowledgments
We thank the anonymous reviewers and our shepherd Isil Dillig for insightful comments. We thank Rajeev Alur, Peter O’Hearn, and Val Tannen for useful feedback. We are grateful to Stephen Freund for helping with FastTrack. This research was supported by DARPA under agreement #FA8750-15-2-0009 and by NSF awards #1253867 and #1526270.

References


