User-Guided Program Reasoning using Bayesian Inference

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

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

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

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

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**;

ACM Reference Format:
Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-Guided Program Reasoning using Bayesian Inference. In *PLDI'18: ACM SIGPLAN Conference on Programming Language Design and Implementation, June 18–22, 2018, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192417

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.

*Co-first authors.

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192417

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 of 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 26 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 `L0` 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 $\mathrm{N}(p_1, p_2)$, $\mathrm{U}(p_1, p_2)$, and $\mathrm{A}(p_1, p_2)$ as input, and produces the relations $\mathrm{P}(p_1, p_2)$ and $\mathrm{race}(p_1, p_2)$ as output. In all relations, variables p_1 and p_2 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, $\mathrm{N}(p_1, p_2)$ may contain the tuples $\mathrm{N}(\mathtt{L1}, \mathtt{L2})$, $\mathrm{N}(\mathtt{L2}, \mathtt{L3})$, etc. While some input relations, such as $\mathrm{N}(p_1, p_2)$, may be directly obtained from the text of the program being analyzed, other input relations, such as $\mathrm{U}(p_1, p_2)$ or $\mathrm{A}(p_1, p_2)$, 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 r_1 may be read as saying, "For all program points p_1, p_2, p_3, if p_1 and p_2 may execute in parallel ($\mathrm{P}(p_1, p_2)$), and p_3 may be executed immediately after p_2 ($\mathrm{N}(p_2, p_3)$), and p_1 and p_3 are not guarded by a common lock ($\mathrm{U}(p_1, p_3)$), then p_1 and p_3 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 $\mathrm{N}(p_1, p_2)$, 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 $\mathrm{P}(\mathtt{L4}, \mathtt{L2})$, 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 $r_1(\mathtt{L4}, \mathtt{L2}, \mathtt{L3})$ indicates the "*grounded instance*" of the rule r_1 with $p_1 = \mathtt{L4}$, $p_2 = \mathtt{L2}$, and $p_3 = \mathtt{L3}$. This clause takes as hypotheses the tuples $\mathrm{P}(\mathtt{L4}, \mathtt{L2})$, $\mathrm{N}(\mathtt{L2}, \mathtt{L3})$, and $\mathrm{U}(\mathtt{L4}, \mathtt{L3})$, and derives the conclusion $\mathrm{P}(\mathtt{L4}, \mathtt{L3})$, and the arrows represent these dependencies.

```java
public class FTPServer {
    public static void main(String args[]) {
        List<RequestHander> conList = new ArrayList<RequestHander>();
        ServerSocket serverSocket = new ServerSocket(...);

        // Start timer thread
        TimerTask timerTask = new TimerTask(conList);
        timerTask.start();

        while (true) {
            Socket socket = serverSocket.accept();
            RequestHander connection = new RequestHandler(socket);
            conList.add(connection);
            // Start connection thread
            connection.start();
        }
    }
}

class TimerTask extends Thread {
    private List<RequestHandler> conList;
    public TimerTask(List<RequestHandler> list) { conList = list; }
    public void run() {
        while (true) {
            for (RequestHandler connection : conList) {
                FtpRequest r = connection.getRequest();
                if (r != null && r.idleTime > ...)
                    connection.close();
            }
        }
    }
}
```

```java
class RequestHandler extends Thread {
    private FtpRequest request;
    private Socket controlSocket;
    private boolean isConnectionClosed = false;

    public RequestHandler (Socket socket) {
        request = new FtpRequest();
        controlSocket = socket;
    }

    public void run() {
        while (...) {
            ... // Do something
        }
        close();
    }

    public FtpRequest getRequest() {
        return request;                         // L0
    }

    public void close() {
        synchronized (this) {                   // L1
            if (isConnectionClosed) return;     // L2
            isConnectionClosed = true;          // L3
        }
        controlSocket.close();                  // L4
        controlSocket = null;                   // L5
        request.clear();                        // L6
        request = null;                         // L7
    }
}
```

Figure 1. Code fragment of an example Java program.

Input relations

$\mathrm{N}(p_1, p_2)$:	Program point p_2 may be executed immediately after program point p_1 by a thread
$\mathrm{U}(p_1, p_2)$:	Program points p_1 and p_2 are not guarded by a common lock
$\mathrm{A}(p_1, p_2)$:	Program points p_1 and p_2 may access the same memory location

Output relations

$\mathrm{P}(p_1, p_2)$:	Program points p_1 and p_2 may be executed by different threads in parallel
$\mathrm{race}(p_1, p_2)$:	Program points p_1 and p_2 may have a datarace

Analysis rules

$$r_1 : \quad \mathrm{P}(p_1, p_3) \coloneq \mathrm{P}(p_1, p_2), \mathrm{N}(p_2, p_3), \mathrm{U}(p_1, p_3).$$
$$r_2 : \quad \mathrm{P}(p_2, p_1) \coloneq \mathrm{P}(p_1, p_2).$$
$$r_3 : \mathrm{race}(p_1, p_2) \coloneq \mathrm{P}(p_1, p_2), \mathrm{A}(p_1, p_2).$$

Figure 2. A simple static datarace analysis in Datalog. We have elided several parts of the analysis, such as the recognition of thread starts and the base case of the $\mathrm{P}(p_1, p_2)$ relation.

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 $\mathrm{P}(\mathtt{L6}, \mathtt{L7})$ can be derived in one of two ways: either by instantiating r_1 with $p_1 = \mathtt{L6}$, $p_2 = \mathtt{L6}$, and $p_3 = \mathtt{L7}$, or by instantiating r_2 with $p_1 = \mathtt{L7}$ and $p_2 = \mathtt{L6}$. 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 $\mathrm{P}(\mathtt{L4}, \mathtt{L2})$, in Figure 3, is true. However, the subsequent conclusion $\mathrm{P}(\mathtt{L4}, \mathtt{L3})$ 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—$\mathrm{race}(\mathtt{L4}, \mathtt{L5})$, $\mathrm{race}(\mathtt{L5}, \mathtt{L5})$, $\mathrm{race}(\mathtt{L6}, \mathtt{L7})$, and $\mathrm{race}(\mathtt{L7}, \mathtt{L7})$—all result from the analysis incorrectly concluding $\mathrm{P}(\mathtt{L4}, \mathtt{L3})$.

2.3 Quantifying incompleteness using probabilities

Incomplete analysis rules are the principal cause of false alarms: even though $\mathrm{P}(\mathtt{L4}, \mathtt{L2})$, $\mathrm{N}(\mathtt{L2}, \mathtt{L3})$ and $\mathrm{U}(\mathtt{L4}, \mathtt{L3})$ are all true, it is not the case that $\mathrm{P}(\mathtt{L4}, \mathtt{L3})$. Bingo addresses this problem by relaxing the interpretation of clause nodes, and only treating them probabilistically:

$$\Pr(r_1(\mathtt{L4}, \mathtt{L2}, \mathtt{L3}) \mid h_1) = 0.95, \text{ and} \tag{1}$$
$$\Pr(\neg r_1(\mathtt{L4}, \mathtt{L2}, \mathtt{L3}) \mid h_1) = 1 - 0.95 = 0.05, \tag{2}$$

where $h_1 = \mathrm{P}(\mathtt{L4}, \mathtt{L2}) \wedge \mathrm{N}(\mathtt{L2}, \mathtt{L3}) \wedge \mathrm{U}(\mathtt{L4}, \mathtt{L3})$ is the event indicating that all the hypotheses of $r_1(\mathtt{L4}, \mathtt{L2}, \mathtt{L3})$ are true, and $p_1 = 0.95$ is the probability of the clause "correctly firing". By setting p_1 to a value strictly less than 1, we make it possible for the conclusion of $r_1(\mathtt{L4}, \mathtt{L2}, \mathtt{L3})$, $\mathrm{P}(\mathtt{L4}, \mathtt{L3})$ to still be false, even though all the hypotheses h_1 hold.

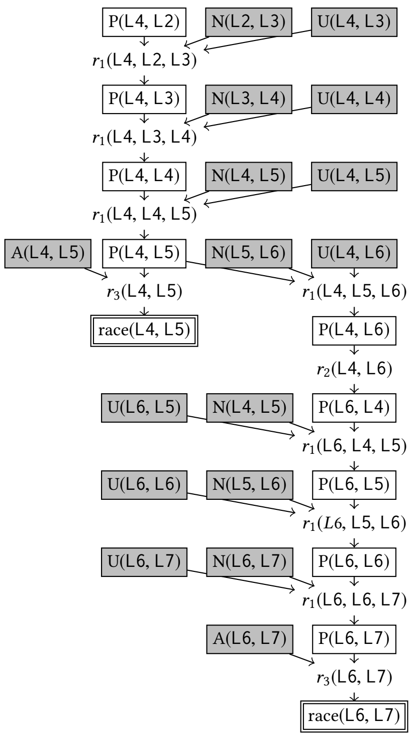

Figure 3. Portion of the derivation graph obtained by applying the static datarace analysis to the program in Figure 1. The central question of this paper is the following: if the user identifies race(L4, L5) as a false alarm, then how should this affect our confidence in the remaining conclusions?

In this new setup, as before, if any of the antecedents of r_1(L4, L2, L3) is false, then it is itself definitely false:

$$\Pr(r_1(\texttt{L4}, \texttt{L2}, \texttt{L3}) \mid \neg h_1) = 0, \text{ and} \tag{3}$$
$$\Pr(\neg r_1(\texttt{L4}, \texttt{L2}, \texttt{L3}) \mid \neg h_1) = 1. \tag{4}$$

We also continue to treat tuple nodes as regular disjunctions:

$$\Pr(\mathrm{P}(\texttt{L6}, \texttt{L7}) \mid r_1(\texttt{L6}, \texttt{L6}, \texttt{L7}) \lor r_2(\texttt{L7}, \texttt{L6})) = 1, \tag{5}$$
$$\Pr(\mathrm{P}(\texttt{L6}, \texttt{L7}) \mid \neg(r_1(\texttt{L6}, \texttt{L6}, \texttt{L7}) \lor r_2(\texttt{L7}, \texttt{L6}))) = 0, \tag{6}$$

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_3 with firing probability $p_3 = 0.95$, and r_2 with probability $p_2 = 1$. Finally, to simplify the discussion, we treat P(L0, L1) and P(L1, L1) as input facts, with $\Pr(\mathrm{P}(\texttt{L0}, \texttt{L1})) = 0.40$ and $\Pr(\mathrm{P}(\texttt{L1}, \texttt{L1})) = 0.60$.

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(L4, L5) as follows:

$$\begin{aligned}
\Pr(\mathrm{race}(\texttt{L4}, \texttt{L5})) &= \Pr(\mathrm{race}(\texttt{L4}, \texttt{L5}) \land r_3(\texttt{L4}, \texttt{L5})) \\
&\quad + \Pr(\mathrm{race}(\texttt{L4}, \texttt{L5}) \land \neg r_3(\texttt{L4}, \texttt{L5})) \\
&= \Pr(\mathrm{race}(\texttt{L4}, \texttt{L5}) \land r_3(\texttt{L4}, \texttt{L5})) \\
&= \Pr(\mathrm{race}(\texttt{L4}, \texttt{L5}) \mid r_3(\texttt{L4}, \texttt{L5})) \cdot \Pr(r_3(\texttt{L4}, \texttt{L5})) \\
&= \Pr(r_3(\texttt{L4}, \texttt{L5}) \mid \mathrm{P}(\texttt{L4}, \texttt{L5}) \land \mathrm{A}(\texttt{L4}, \texttt{L5})) \\
&\quad \cdot \Pr(\mathrm{P}(\texttt{L4}, \texttt{L5})) \cdot \Pr(\mathrm{A}(\texttt{L4}, \texttt{L5})) \\
&= 0.95 \cdot \Pr(\mathrm{P}(\texttt{L4}, \texttt{L5})) = 0.95^4 \cdot \Pr(\mathrm{P}(\texttt{L4}, \texttt{L2})) \\
&= 0.95^8 \cdot \Pr(\mathrm{P}(\texttt{L1}, \texttt{L1})) = 0.398.
\end{aligned}$$

The user now inspects the top-ranked report, race(L4, L5), 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, $\Pr(a \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5}))$, 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(L6, L7) drops from 0.324 to 0.030, while the belief in the unrelated alarm race(L0, L7) 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:

$$\begin{aligned}
&\Pr(\mathrm{race}(\texttt{L6}, \texttt{L7}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) \\
&\quad = \Pr(\mathrm{race}(\texttt{L6}, \texttt{L7}) \land \mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) \\
&\qquad + \Pr(\mathrm{race}(\texttt{L6}, \texttt{L7}) \land \neg\,\mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) \\
&\quad = \Pr(\mathrm{race}(\texttt{L6}, \texttt{L7}) \land \mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})).
\end{aligned}$$

Next, race(L4, L5) and race(L6, L7) are conditionally independent given P(L4, L5) as it occurs on the unique path between them. So,

$$\begin{aligned}
&\Pr(\mathrm{race}(\texttt{L6}, \texttt{L7}) \land \mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) \\
&= \Pr(\mathrm{race}(\texttt{L6}, \texttt{L7}) \mid \mathrm{P}(\texttt{L4}, \texttt{L5})) \cdot \Pr(\mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) \\
&= 0.95^5 \cdot \Pr(\mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})).
\end{aligned}$$

Finally, by Bayes' rule, we have:

$$\begin{aligned}
&\Pr(\mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) \\
&\quad = \frac{\Pr(\neg\,\mathrm{race}(\texttt{L4}, \texttt{L5}) \mid \mathrm{P}(\texttt{L4}, \texttt{L5})) \cdot \Pr(\mathrm{P}(\texttt{L4}, \texttt{L5}))}{\Pr(\neg\,\mathrm{race}(\texttt{L4}, \texttt{L5}))} \\
&\quad = \frac{0.05 \cdot 0.95^7 \cdot 0.60}{0.60} = 0.03.
\end{aligned}$$

Our prior belief in P(L4, L5) was $\Pr(\mathrm{P}(\texttt{L4}, \texttt{L5})) = 0.42$, so that $\Pr(\mathrm{P}(\texttt{L4}, \texttt{L5}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) \ll \Pr(\mathrm{P}(\texttt{L4}, \texttt{L5}))$, but is still strictly greater than 0. This is because one eventuality by which $\neg$ race(L4, L5) may occur is for P(L4, L5) to be true, but for the clause r_3(L4, L5) to misfire. We may now conclude that $\Pr(\mathrm{race}(\texttt{L6}, \texttt{L7}) \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})) = 0.95^5 \cdot 0.03 = 0.030$.

Rank	Belief	Program points
1	0.398	`RequestHandler : L4`, `RequestHandler : L5`
2	0.378	`RequestHandler : L5`, `RequestHandler : L5`
3	0.324	`RequestHandler : L6`, `RequestHandler : L7`
4	0.308	`RequestHandler : L7`, `RequestHandler : L7`
5	0.279	`RequestHandler : L0`, `RequestHandler : L7`

(a) $\Pr(a)$.

Rank	Belief	Program points
1	0.279	`RequestHandler : L0`, `RequestHandler : L7`
2	0.035	`RequestHandler : L5`, `RequestHandler : L5`
3	0.030	`RequestHandler : L6`, `RequestHandler : L7`
4	0.028	`RequestHandler : L7`, `RequestHandler : L7`
5	0	`RequestHandler : L4`, `RequestHandler : L5`

(b) $\Pr(a \mid \neg \text{race}(20, 21))$.

Table 1. List of alarms produced by Bingo, **(a)** before, and **(b)** after the feedback ¬ race(`L4`, `L5`). Observe how the real datarace race(`L0`, `L7`) rises in the ranking as a result of feedback.

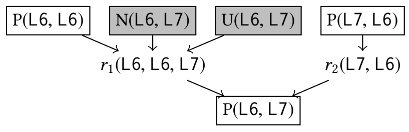

Figure 4. Another portion of the derivation graph showing multiple ways of deriving the tuple P(`L6`, `L7`).

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.

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].

Algorithm 1 Bingo$(D, P, \boldsymbol{p})$, where D is the analysis expressed in Datalog, P is the program to be analyzed, and $\boldsymbol{p}$ 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 $(C, A, GC) = \text{DatalogSolve}(D, I)$. $C$ 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 $GC_c := \text{CycleElim}(I, GC)$. Eliminate cycles
   from the grounded constraints.
4. (Optionally,) Update $GC_c := \text{Optimize}(I, GC_c, A)$. Reduce
   the size of the set of grounded constraints.
5. Construct Bayesian network $BN$ from $GC_c$ and $\boldsymbol{p}$, and let
   $\Pr$ be its joint probability distribution.
6. Initialize the feedback set $\boldsymbol{e} := \emptyset$.
7. While there exists an unlabelled alarm:
   a. Let $A_u = A \setminus \boldsymbol{e}$ be the set of all unlabelled alarms.
   b. Determine the top-ranked unlabelled alarm:
          $a_t = \arg\max_{a \in A_u} \Pr(a \mid \boldsymbol{e})$.
   c. Present $a_t$ for inspection. If the user labels it a true alarm,
      update $\boldsymbol{e} := \boldsymbol{e} \cup \{a_t\}$. Otherwise, update $\boldsymbol{e} := \boldsymbol{e} \cup \{\neg a_t\}$.
```

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:

$$R_h(\boldsymbol{u}_h) :- R_1(\boldsymbol{u}_1), R_2(\boldsymbol{u}_2), \ldots, R_p(\boldsymbol{u}_p),$$

where R_h is an output relation, and $\boldsymbol{u}_h$, $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, ..., $\boldsymbol{u}_p$ 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 $\boldsymbol{u}$ of the free variables, if $R_1(\boldsymbol{u}_1)$, and $R_2(\boldsymbol{u}_2)$, ..., and $R_p(\boldsymbol{u}_p)$, then $R_h(\boldsymbol{u}_h)$".

Instantiating the free variables of a rule yields a Horn clause, $R_1(\boldsymbol{v}_1) \wedge R_2(\boldsymbol{v}_2) \wedge \cdots \wedge R_p(\boldsymbol{v}_p) \implies R_h(\boldsymbol{v}_h)$. For example, the constraint r_1(`L4`, `L2`, `L3`) from Figure 3 represents the Horn clause P(`L4`, `L2`) ∧ N(`L2`, `L3`) ∧ U(`L4`, `L3`) $\implies$ P(`L4`, `L3`).

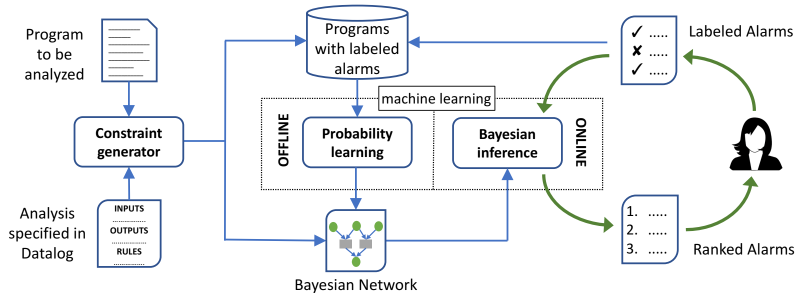

Figure 5. The Bingo workflow and interaction model.

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, $C := I$, and initialize the grounded constraints to the set of input tuples, $GC := \{\text{True} \implies t \mid t \in I\}$. We repeatedly apply each rule to update C and GC until no new conclusions can be reached. That is, whenever $R_1(\boldsymbol{v}_1), R_2(\boldsymbol{v}_2), \ldots, R_p(\boldsymbol{v}_p)$ occur in C, we update $C := C \cup \{R_h(\boldsymbol{v}_h)\}$, and $GC := GC \cup \{R_1(\boldsymbol{v}_1) \wedge R_2(\boldsymbol{v}_2) \wedge \cdots \wedge R_p(\boldsymbol{v}_p) \implies R_h(\boldsymbol{u}_h)\}$.

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* $\mathcal{G} = (V, E)$ 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,

$$\text{Pa}(v) = \{u \in V \mid u \to v \in E\}.$$

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

$$\Pr(\boldsymbol{x}) = \prod_v p(x_v \mid \boldsymbol{x}_{\text{Pa}(v)}), \tag{7}$$

where the joint assignment $\boldsymbol{x}$ is a valuation x_v for each $v \in V$, and $\boldsymbol{x}_{\text{Pa}(v)}$ 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, $\Pr(v) = \sum_{\{\boldsymbol{x} \mid x_v = \text{True}\}} \Pr(\boldsymbol{x})$, $\Pr(\neg v) = 1 - \Pr(v)$, and the conditional probability of arbitrary events: $\Pr(v \mid \boldsymbol{e}) = \Pr(v \wedge \boldsymbol{e}) / \Pr(\boldsymbol{e})$.

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 $\mathcal{G}(C, GC)$ 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.

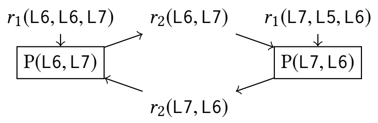
$r_1(\text{L6}, \text{L6}, \text{L7})$
$r_2(\text{L6}, \text{L7})$
$r_1(\text{L7}, \text{L5}, \text{L6})$
$\text{P}(\text{L6}, \text{L7})$
$\text{P}(\text{L7}, \text{L6})$
$r_2(\text{L7}, \text{L6})$

Figure 6. An example of a cycle in the derivation graph.

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 $\mathcal{G}(C, GC_c)$ 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 $\text{CycleElim}(I, GC)$ satisfies the desired requirements:

Theorem 3.1. *For all I and GC, if $GC_c = \text{CycleElim}(I, GC)$, then (a) $GC_c \subseteq GC$, (b) every tuple derivable using GC is also derivable using GC_c, and (c) $\mathcal{G}(C, GC_c)$ is acyclic.*

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, $\boldsymbol{p}$. To emphasize this dependence, we write $\Pr_{\boldsymbol{p}}$ for the induced joint probability distribution.

We will now discuss how we may obtain $\boldsymbol{p}$. 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

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 $t \in I$, $T(t) = 0$, and otherwise, $T(t) = \infty$.
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
$$T(c_g) > \max_{a \in A_g}(T(a)) + 1, \text{ update:}$$
$$T(c_g) := \max_{a \in A_g}(T(a)) + 1. \tag{8}$$
4. Define $GC_c = \{g \in GC \mid T(c_g) > \max_{a \in A_g}(T(a))\}$. $GC_c \subseteq GC$ is the set of all those constraints in GC whose consequent has a timestamp *strictly greater* than all of its antecedents. Return GC_c.

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 $\boldsymbol{v}$, our learning problem is to determine the "best" weight vector $\boldsymbol{p}$ which explains $\boldsymbol{v}$. One common measure of goodness is the *likelihood*, $L(\boldsymbol{p}; \boldsymbol{v})$, which may be informally motivated as the plausibility of $\boldsymbol{p}$ given the training data $\boldsymbol{v}$. Concretely, $L(\boldsymbol{p}; \boldsymbol{v}) = \Pr_{\boldsymbol{p}}(\boldsymbol{v})$. To "learn" the rule firing probabilities is to then find the probability vector $\tilde{\boldsymbol{p}}$ with highest likelihood:

$$\tilde{\boldsymbol{p}} = \arg\max_{\boldsymbol{p}} L(\boldsymbol{p}; \boldsymbol{v}). \tag{9}$$

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 $\boldsymbol{p}^{(0)}$, the algorithm iteratively computes a sequence of estimates $\boldsymbol{p}^{(0)}, \boldsymbol{p}^{(1)}, \boldsymbol{p}^{(2)}, \ldots$, as follows:

$$p_r^{(t+1)} = \frac{\sum_{g_r} \Pr_{\boldsymbol{p}^{(t)}}(c_{g_r} \wedge A_{g_r})}{\sum_{g_r} \Pr_{\boldsymbol{p}^{(t)}}(A_{g_r})}, \tag{10}$$

where g_r ranges over all grounded clauses associated with r. The algorithm guarantees that $L(\boldsymbol{p}^{(t)}; \boldsymbol{v})$ monotonically increases with t, 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. $i < j \wedge \neg v_i \wedge v_j$. 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 wish to choose the ranking algorithm with lowest expected inversion count. There are two versions of this problem which are relevant in our setting: (a) ranking the alarms given a fixed set of observations $\boldsymbol{e}$, and (b) ranking the alarms in an interactive setting, where $\boldsymbol{e}$ grows with each iteration. The following theorem states that Bingo-style ranking is optimal for the first problem, and we present the proof in Appendix A.

Theorem 3.2. *For each set of observations $\boldsymbol{e}$, the sequence $w = a_1, a_2, \ldots, a_n$ of alarms, arranged according to decreasing $\Pr(a_i \mid \boldsymbol{e})$, 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. Bingo 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 Bingo is the conditional probability query $\Pr(a \mid \boldsymbol{e})$ 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 $\mathsf{P}(\mathtt{L4}, \mathtt{L2}) \longrightarrow \mathsf{P}(\mathtt{L4}, \mathtt{L3}) \longrightarrow \mathsf{P}(\mathtt{L4}, \mathtt{L4}) \longrightarrow \mathsf{P}(\mathtt{L4}, \mathtt{L5})$. 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) ⟶ 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:
       $\textsc{Srcs}(t) := \{g \in GC \mid t = c_g\}$,
       $\textsc{Sinks}(t) := \{g \in GC \mid t \in A_g\}$.
2. Construct the following set:
       $E := \{t \in C \setminus A \mid |\textsc{Srcs}(t)| = 1 \wedge |\textsc{Sinks}(t)| = 1\}$.
3. While $E$ is not empty:
    a. Pick an arbitrary tuple $t \in E$, and let $\textsc{Srcs}(t) = \{g_1\}$,
       and $\textsc{Sinks}(t) = \{g_2\}$.
    b. Since $t = c_{g_1}$ and $t$ s an antecedent of $g_2$, let
           $g_1 = a_1 \wedge a_2 \wedge \cdots \wedge a_k \implies t$, and
           $g_2 = t \wedge b_1 \wedge b_2 \wedge \ldots b_p \implies t'$.
    c. Define a new clause, $g' = a_1 \wedge a_2 \wedge \cdots \wedge a_k \wedge b_1 \wedge b_2 \wedge$
       $\cdots \wedge b_p \implies t'$. Update $GC := GC \cup \{g'\} \setminus \{g_1, g_2\}$,
       $E := E \setminus \{t\}$, and recompute Srcs and Sinks.
    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 LibDAI version 0.3.2. We set a timeout of 2 hours per alarm proposed for both Bingo and the baselines.

Analysis	Rules	Input relations	Output relations
Datarace	102	58	44
Taint	62	52	25

Table 2. Statistics of the instance analyses.

5.1.1 Instance analyses

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

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 *soundy* [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 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

	Program	Description	# Classes		# Methods		Bytecode (KLOC)	
			Total	App	Total	App	Total	App
Datarace analysis	`hedc`	Web crawler from ETH	357	44	2,154	230	141	11
	`ftp`	Apache FTP server	499	119	2,754	608	152	23
	`weblech`	Website download/mirror tool	579	56	3,344	303	167	12
	`jspider`	Web spider engine	362	113	1,584	426	95	13
	`avrora`	AVR microcontroller simulator	2,080	1,119	10,095	3,875	369	113
	`luindex`	Document indexing tool	1,168	169	7,494	1,030	317	47
	`sunflow`	Photo-realistic image rendering system	1,857	127	12,934	967	616	53
	`xalan`	XML to HTML transforming tool	1,727	390	12,214	3,007	520	120
Taint analysis	`app-324`	Unendorsed Adobe Flash player	1,788	81	6,518	167	40	10
	`noisy-sounds`	Music player	1,418	119	4,323	500	52	11
	`app-ca7`	Simulation game	1,470	142	4,928	889	55	23
	`app-kQm`	Puzzle game	1,332	105	4,114	517	68	31
	`tilt-mazes`	Game packaging the Mobishooter malware	2,462	547	7,034	2,815	77	35
	`andors-trail`	RPG game infected with malware	1,623	339	5,016	1,523	81	44
	`ginger-master`	Image processing tool	1,474	159	4,500	738	82	39
	`app-018`	Arcade game	1,840	275	5,397	1,389	98	50

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.

the benchmarks, `hedc`, `ftp`, `weblech`, and `jspider`. 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 `luindex` 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 and 587 for Base-C and Base-R respectively. 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 *ROC curve* [14] as another representation of the interaction process. We present 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 in the process 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 χ 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 analysis 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 sophisticated dynamic datarace detector based on the happens-before relation. We ran FastTrack with the inputs that were supplied with the benchmarks.

	Program	#Alarms			Rank-100%-T			Rank-90%-T			Area under the curve (AUC)		
		Total	Bugs	%TP	Bingo	Base-C	Base-R	Bingo	Base-C	Base-R	Bingo	Base-C	Base-R
Datarace analysis	`hedc`	152	12	7.89%	**67**	121	143	**65**	115	135	**0.81**	0.76	0.50
	`ftp`	522	75	14.37%	**103**	368	520	**80**	290	476	**0.98**	0.78	0.49
	`weblech`	30	6	20.00%	**11**	16	29	**10**	15	25	**0.84**	0.78	0.48
	`jspider`	257	9	3.50%	**20**	128	247	**19**	101	201	**0.97**	0.81	0.59
	`avrora`	978	29	2.97%	**410**	971	960	**365**	798	835	**0.75**	0.70	0.51
	`luindex`	940	2	0.21%	**14**	101	587	**14**	101	587	**0.99**	0.89	0.61
	`sunflow`	958	171	17.85%	**838**	timeout	952	**483**	timeout	872	**0.79**	timeout	0.50
	`xalan`	1,870	75	4.01%	**273**	timeout	1844	**266**	timeout	1,706	**0.91**	timeout	0.50
Taint analysis	`app-324`	110	15	13.64%	**51**	104	106	**44**	89	97	**0.83**	0.58	0.50
	`noisy-sounds`	212	52	24.53%	**135**	159	207	**79**	132	190	**0.89**	0.69	0.50
	`app-ca7`	393	157	39.95%	**206**	277	391	**172**	212	350	**0.96**	0.81	0.51
	`app-kQm`	817	160	19.58%	**255**	386	815	**200**	297	717	**0.93**	0.86	0.51
	`tilt-mazes`	352	150	42.61%	**221**	305	351	**155**	205	318	**0.95**	0.79	0.50
	`andors-trail`	156	7	4.49%	**14**	48	117	**13**	44	92	**0.98**	0.81	0.60
	`ginger-master`	437	87	19.91%	**267**	303	436	**150**	214	401	**0.84**	0.77	0.47
	`app-018`	420	46	10.95%	**288**	311	412	**146**	186	369	**0.85**	0.77	0.51

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.

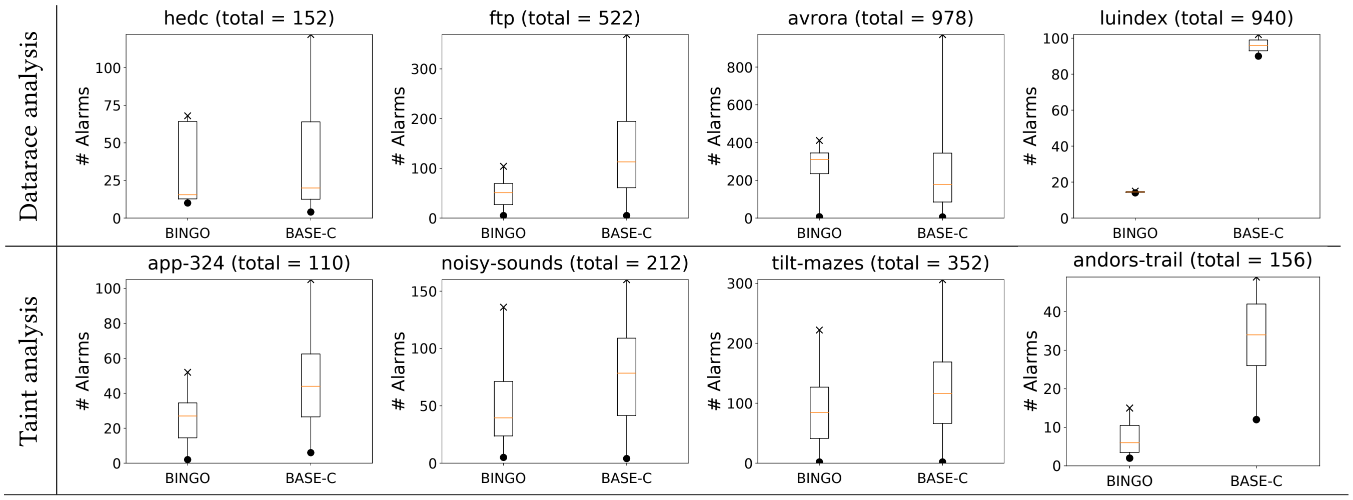

Figure 7. Comparing the interaction runs produced by Bingo and Base-C. The y-axis shows the number of alarms inspected by the user, the "●" and "×" indicate the rounds in which the first and last true alarms were discovered, and the boxes indicate the rounds in which 25%, 50%, and 75% of the true alarms were discovered. Each measurement for Base-C is itself the median of 5 independent runs. The plots for the remaining benchmarks are available in Appendix C.

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.

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 dataraces are notoriously hard to diagnose. However, in the user study described in

Program	Chord, soundy		Chord, unsound			Missed by	Bingo	
	Total	Bugs	Total	Bugs	Missed	FastTrack	New bugs	LTR
`hedc`	152	12	55	6	6	5	3	67
`ftp`	522	75	211	36	39	29	14	103
`weblech`	30	6	7	4	2	0	0	11
`jspider`	257	9	52	5	4	2	0	20
`avrora`	978	29	9	4	25	7	6	410
`luindex`	940	2	494	2	0	1	0	14
`sunflow`	958	171	506	94	77	151	69	838
`xalan`	1,870	75	80	52	23	8	8	273
Total	**5,707**	**379**	**1,414**	**203**	**176**	**203**	**100**	**1,736**

Table 5. The number of real dataraces missed by Chord's datarace analysis with unsound settings, and the FastTrack dynamic datarace detector, for 8 Java benchmarks. New bugs are the real dataraces proposed by Bingo but missed by both. LTR is the rank at which Bingo discovers all true alarms.

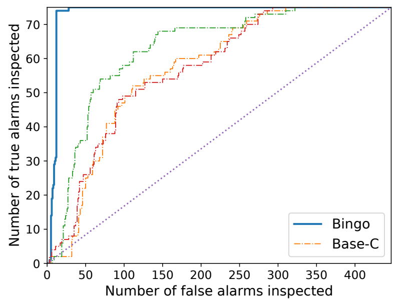

Figure 8. The ROC curves for `ftp`. The solid line is the curve for Bingo, while the dotted lines are the curves for each of the runs of the Base-C.

	Tool	Rank-100%-T	Rank-90%-T	AUC
Exact	Bingo	103	80	0.98
	Base-C	368	290	0.78
Noisy	Bingo (1% noise)	111	85	0.97
	Bingo (5% noise)	128	88	0.93
	Bingo (10% noise)	203	98	0.86

Table 6. Robustness of Bingo with varying amounts of user error in labelling alarms for the `ftp` benchmark. Each value is the median of three measurements.

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 from 103 in the original setting to 203 in the presence of noise, but the rank at which 90% of the true alarms have 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

Program	Bingo, optimized			Bingo, unoptimized			Base-C		
	#Tuples (K)	#Clauses (K)	Iter time (s)	#Tuples (K)	#Clauses (K)	Iter time (s)	#Vars (K)	#Clauses (K)	Iter time (s)
`hedc`	12	10	46	753	789	12,689	1,298	1,468	194
`ftp`	111	112	1,341	2,067	2,182	37,447	2,859	3,470	559
`weblech`	2	1	3	497	524	7,950	1,498	1,718	290
`jspider`	45	45	570	1,126	1,188	12,982	1,507	1,858	240
`avrora`	56	53	649	1,552	1,824	47,552	2,305	3,007	1,094
`luindex`	35	22	41	488	522	9,334	1,584	1,834	379
`sunflow`	277	290	3,636	9,632	11,098	timeout	26,025	34,218	timeout
`xalan`	101	92	489	2,452	2,917	51,812	6,418	8,660	timeout

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.

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 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

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1994. *Foundations of databases: The logical level* (1st ed.). Pearson.

[2] Michael Arntzenius and Neelakantan Krishnaswami. 2016. Datafun: A functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, 214–227.

[3] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2:1–2:25.

[4] Stephen Blackburn, Robin Garner, Chris Hoffmann, Asjad Khang, Kathryn McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Guyer, Martin Hirzel, Anthony Hosking, Maria Jump, Han Lee, Eliot Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*. ACM, 169–190.

[5] Sam Blackshear and Shuvendu Lahiri. 2013. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM, 209–218.

[6] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. ACM, 241–250.

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2009)*. ACM, 243–262.

[8] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically generating features for learning program analysis heuristics for C-like languages. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 101 (Oct. 2017), 25 pages.

[9] Supratik Chakraborty, Daniel Fremont, Kuldeep Meel, Sanjit Seshia, and Moshe Vardi. 2014. Distribution-aware sampling and weighted model counting for SAT. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*. AAAI Press, 1722–1730.

[10] Andy Chou. 2014. On detecting Heartbleed with static analysis. https://www.synopsys.com/blogs/software-security/detecting-heartbleed-with-static-analysis/. (2014).

[11] Nilesh Dalvi and Dan Suciu. 2004. Efficient query evaluation on probabilistic databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*. VLDB Endowment, 864–875.

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM, 181–192.

[13] Mahdi Eslamimehr and Jens Palsberg. 2014. Race directed scheduling of concurrent programs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2014)*. ACM, 301–314.

[14] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern Recognition Letters* 27, 8 (2006), 861 – 874.

[15] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 576–587.

[16] Dan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming* 15, 3 (2015), 358–401.

[17] Cormac Flanagan and Stephen Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*. ACM, 121–133.

[18] Norbert Fuhr. 1995. Probabilistic Datalog: A logic for powerful retrieval methods. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1995)*. ACM, 282–290.

[19] Michael Garey and David Johnson. 1979. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman.

[20] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *28th International Conference on Computer Aided Verification (CAV 2016)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 62–83.

[21] Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Ben Taskar. 2007. Probabilistic relational models. In *Introduction to Statistical Relational Learning*, Lise Getoor and Ben Taskar (Eds.). MIT Press, 129–174.

[22] Patrice Godefroid. 2005. The soundness of bugs is what matters. In *Proceedings of BUGS 2005*.

[23] Radu Grigore and Hongseok Yang. 2016. Abstraction refinement guided by a learnt probabilistic model. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, 485–498.

[24] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. IEEE Press, 519–529.

[25] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. 2005. Taming false alarms from a domain-unaware C Analyzer by a Bayesian statistical post analysis. In *Static Analysis: 12th International Symposium (SAS 2005)*, Chris Hankin and Igor Siveroni (Eds.). Springer, 203–217.

[26] Kristian Kersting and Luc De Raedt. 2007. Bayesian logic programming: Theory and tool. In *Introduction to Statistical Relational Learning*, Lise Getoor and Ben Taskar (Eds.). MIT Press, 291–322.

[27] Davis King. 2009. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research* 10 (2009), 1755–1758.

[28] Ugur Koc, Parsa Saadatpanah, Jeffrey Foster, and Adam Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. ACM, 35–42.

[29] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: Principles and techniques*. The MIT Press.

[30] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation exploitation in error ranking. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2004/FSE-12)*. ACM, 83–93.

[31] Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis: 10th International Symposium (SAS 2003)*, Radhia Cousot (Ed.). Springer, 295–315.

[32] Ted Kremenek, Andrew Ng, and Dawson Engler. 2007. A factor graph model for software bug finding. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI 2007)*. Morgan Kaufmann, 2510–2516.

[33] Frank Kschischang, Brendan Frey, and Hans-Andrea Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on*

Information Theory 47, 2 (Feb 2001), 498–519.

[34] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. 2016. Accelerating program analyses by cross-program training. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, 359–377.

[35] Wei Le and Mary Lou Soffa. 2010. Path-based fault correlations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010)*. ACM, 307–316.

[36] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012. Sound non-statistical clustering of static analysis alarms. In *Verification, Model Checking, and Abstract Interpretation: 13th International Conference (VMCAI 2012)*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer, 299–314.

[37] Benjamin Livshits and Shuvendu Lahiri. 2014. In defense of probabilistic analysis. In *1st SIGPLAN Workshop on Probabilistic and Approximate Computing*.

[38] Benjamin Livshits, Aditya Nori, Sriram Rajamani, and Anindya Banerjee. 2009. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*. ACM, 75–86.

[39] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: A manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46.

[40] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. ACM, 194–208.

[41] Ravi Mangal, Xin Zhang, Aditya Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 462–473.

[42] Ana Milanova, Atanas Rountev, and Barbara Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1 (Jan. 2005), 1–41.

[43] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*. Morgan Kaufmann, 1352–1359.

[44] Thomas Minka, John Winn, John Guiver, Sam Webster, Yordan Zaykov, Boris Yangel, Alexander Spengler, and John Bronskill. 2014. Infer.NET 2.6. (2014). Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[45] Joris Mooij. 2010. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research* 11 (Aug 2010), 2169–2173.

[46] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 151–162.

[47] Kevin Murphy, Yair Weiss, and Michael Jordan. 1999. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the 15th Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI 1999)*. Morgan Kaufmann, 467–476.

[48] Mayur Naik. 2006. Chord: A program analysis platform for Java. http://jchord.googlecode.com/. (2006).

[49] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*. ACM, 308–319.

[50] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. 2011. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *Proceedings of the VLDB Endowment* 4, 6 (March 2011), 373–384.

[51] Oded Padon, Kenneth McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. ACM, 614–630.

[52] Judea Pearl. 1985. *Bayesian networks: A model of self-activated memory for evidential reasoning*. Technical Report CSD-850017. University of California Los Angeles.

[53] Judea Pearl. 1988. *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. Morgan Kaufmann.

[54] Martin Puterman. 1994. *Markov decision processes: Discrete stochastic dynamic programming* (1st ed.). Wiley.

[55] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. ACM, 111–124.

[56] Thomas Reps. 1995. *Demand interprocedural program analysis using logic databases*. Springer, 163–196.

[57] Matthew Richardson and Pedro Domingos. 2006. Markov logic networks. *Machine Learning* 62, 1 (01 Feb 2006), 107–136.

[58] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*. ACM, 762–774.

[59] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot: A Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*. IBM Press.

[60] John Whaley, Dzintars Avots, Michael Carbin, and Monica Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems: Third Asian Symposium. Proceedings*, Kwangkeun Yi (Ed.). Springer, 97–118.

[61] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 57 (Oct. 2017), 30 pages.

A Framework for Bayesian Inference

We will now prove the two theorems stated in Section 3.

A.1 Correctness of cycle elimination

Theorem 3.1. *For all I and GC, if $GC_c = \textsc{CycleElim}(I, GC)$, then (a) $GC_c \subseteq GC$, (b) every tuple derivable using GC is also derivable using GC_c, and (c) $\mathcal{G}(C, GC_c)$ is acyclic.*

Proof. The first part of the claim is immediate because of the definition of GC_c in Algorithm 2.

We will now prove the second part of the claim. If t is an input tuple, then the result is immediate, because of the presence of the clause True $\implies t$ in GC_c. We will now focus our attention on derivable non-input tuples t. At the fixpoint of step 3 of the algorithm, observe that every derivable tuple t has a finite-valued timestamp, $T(t) \lneq \infty$. We prove the derivability of t in GC_c by induction on its timestamp $T(t)$. We already know the result for the case of $T(t) = 0$, as t is then an input tuple. If $T(t) = n + 1$, and assuming the result for all tuples t' with $T(t') \leq n$, consider the deriving clause g_t, and observe that each of its antecedents $t'' \in A_{g_t}$ has a small timestamp, $T(t'') \leq n$. By the induction hypothesis, all these tuples t'' are derivable, and it follows that t is itself derivable within GC_c.

Finally, observe that for every clause $g \in GC_c$, the timestamp of the consequent $T(c_g)$ is strictly greater than the timestamp of each of its antecedents, $T(a)$, for $a \in A_g$. This rules out the possibility of a cycle in GC_c. □

A.2 The inversion count, and optimality of ranking

Let $w = a_1, a_2, \ldots, a_n$ be a sequence of alarms. If n_t and n_f are the number of true and false alarms in w, then observe that for a perfect ranking w_g, with all true alarms before all false positives, $\chi(w_g) = 0$, whereas an ordering w_b with all false positives before all true alarms, has $\chi(w_b) = n_f n_t$.

Next, if $\Pr(\cdot)$ describes the joint probability distribution of alarms, observe that the expected inversion count of $w = a_1, a_2, \ldots, a_n$ of alarms is given by:

$$\mathrm{E}(\chi(w) \mid \boldsymbol{e}) = \sum_i \sum_{j>i} \Pr(\neg a_i \wedge a_j \mid \boldsymbol{e}). \tag{11}$$

We can now prove Theorem 3.2.

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

Proof. We will first prove the theorem for the case with $n = 2$ alarms, and then generalize to larger values of n.

***Case 1** ($n = 2$).* There are exactly two ways of arranging a pair of alarms: $w = a_1, a_2$, and $w' = a_2, a_1$, with expected inversion counts

$$\mathrm{E}(\chi(w) \mid \boldsymbol{e}) = \Pr(\neg a_1 \wedge a_2 \mid \boldsymbol{e}), \text{ and}$$
$$\mathrm{E}(\chi(w') \mid \boldsymbol{e}) = \Pr(a_1 \wedge \neg a_2 \mid \boldsymbol{e})$$

respectively. Our hypothesis states that

$$\Pr(a_1 \mid \boldsymbol{e}) \geq \Pr(a_2 \mid \boldsymbol{e}).$$

Rewriting each of these events as the union of a pair of mutually exclusive events, we have:

$$\Pr(a_1 \wedge a_2 \mid \boldsymbol{e}) + \Pr(a_1 \wedge \neg a_2 \mid \boldsymbol{e}) \geq \Pr(a_1 \wedge a_2 \mid \boldsymbol{e}) + \Pr(\neg a_1 \wedge a_2 \mid \boldsymbol{e}),$$

so that $\mathrm{E}(\chi(w' \mid \boldsymbol{e})) \geq \mathrm{E}(\chi(w \mid \boldsymbol{e}))$, thus establishing our result for the case when $n = 2$.

***Case 2** ($n > 2$).* We will now prove the result for larger values of n by piggy-backing on bubble sort. For the sake of contradiction, let us assume that some other sequence $w' = a'_1, a'_2, \ldots, a'_n$ has lower expected inversion count,

$$\mathrm{E}(\chi(w' \mid \boldsymbol{e})) < \mathrm{E}(\chi(w) \mid \boldsymbol{e}). \tag{12}$$

Associate each alarm a'_i with its index j in the reference ordering w, so that $a'_i = a_j$, and run bubble sort on w' according to the newly associated keys. For each k, let $w^{(k)}$ be the state of the sequence after the sorting algorithm has swapped k elements, so that we end up with a sequence of orderings, $w^{(0)}, w^{(1)}, w^{(2)}, \ldots, w^{(m)}$, such that $w' = w^{(0)}$ and $w^{(m)} = w$.

Now consider each pair of consecutive sequences, $w^{(k)}$ and $w^{(k+1)}$. Except for a pair of adjacent elements, $a_i^{(k)}, a_{i+1}^{(k)}$, the two sequence have the same alarms at all other locations. Because they were swapped, it follows that $a_{i+1}^{(k)}$ appears before $a_i^{(k)}$ in $w^{(m)} = w$, so that

$$\Pr(a_{i+1}^{(k)} \mid \boldsymbol{e}) \geq \Pr(a_i^{(k)} \mid \boldsymbol{e}).$$

From Equation 11 and by an argument similar to that used in the $n = 2$ case, we conclude that $\mathrm{E}(\chi(w^{(k)}) \mid \boldsymbol{e}) \geq \mathrm{E}(\chi(w^{(k+1)}) \mid \boldsymbol{e})$, and transitively that $\mathrm{E}(\chi(w^{(0)}) \mid \boldsymbol{e}) \geq \mathrm{E}(\chi(w^{(m)}) \mid \boldsymbol{e})$. Since $w' = w^{(0)}$ and $w^{(m)} = w$, this immediately conflicts with our assumption that $\mathrm{E}(\chi(w' \mid \boldsymbol{e})) < \mathrm{E}(\chi(w) \mid \boldsymbol{e})$, and completes the proof. □

A.3 Comparison to probabilistic Datalog

One prominent distinction between probabilistic databases and our model is in the source of uncertainties. While we assume that the input tuples are known with surety and that uncertainty in conclusions arises from *incomplete rules*, the literature on probabilistic databases mostly assumes that the rules are inerrant and that the presence / absence of individual input tuples is what is uncertain. As a result, it is non-trivial to translate problem instances from our setting to probabilistic Datalog.

B Implementation

Engineering the belief propagation algorithm. In each round of the inference algorithm, each random variable v passes a message to all its neighbors u (both parents and children), indicating the updated belief in u given the current belief in v. Each node then computes a new belief by combining the messages received from all its neighbors. The algorithm terminates when the beliefs converge, i.e. they do not differ by more than ϵ from one round to the next.

We were motivated by the predictable, deterministic behavior of this algorithm, and because it scales well to the graphs produced by our benchmarks. Furthermore, even though it is only an approximate inference algorithm for general Bayesian networks, empirical studies show that it often converges to values that are very close to the true marginal probabilities [47].

Unfortunately, as reported by previous studies [47], and as we observed in our early experiments, loopy BP sometimes does not converge, and the beliefs occasionally exhibit oscillating behavior. Whenever this phenomenon occurred, we observed that it is only a small subset of variables exhibiting pathological behavior. We overcame this difficulty in two ways: (*a*) by setting a timeout on the number of iterations that the algorithm can execute, and terminating even if the beliefs of some nodes had not yet converged, and (*b*) in case of non-convergence, by returning the average belief over the last 100 iterations, instead of simply the final value, as a low-pass filter to dampen any oscillatory behavior.

C Experimental Evaluation

The baseline ranker, Base-C. We constructed Base-C by repurposing the alarm classifier Eugene [41]. Let A be the set of alarms produced by the analysis. Given positive and negative feedback on some disjoint subsets of alarms, $L_p, L_n \subseteq A$, Eugene generalizes this and classifies *all* $a \in A$ as likely true and likely false: A_p, A_n, such that $A_p \cup A_n = A$, $L_p \subseteq A_p$, and $L_n \subseteq A_n$. The natural way to make this system user-guided is the following algorithm, which we implemented as Base-C:

1. Initialize L_p and L_n to $\emptyset$.
2. While there exists an unlabelled alarm:
   a. Invoke the classifier: $(A_p, A_n) = \textsc{Classify}(L_p, L_n)$.
   b. Pick an unlabelled alarm a uniformly at random from A_p. If no such alarm exists, choose it uniformly at random from A_n.
   c. Present a for inspection by the user, and insert it into L_p or L_n as appropriate.

Supplemental data. Figure 9 shows the comparison between Bingo and Base-C for the benchmark programs we omitted from Figure 7. Table 8 supplements Table 7, and reports the effect of our optimizations on the execution time for the taint analysis benchmarks.

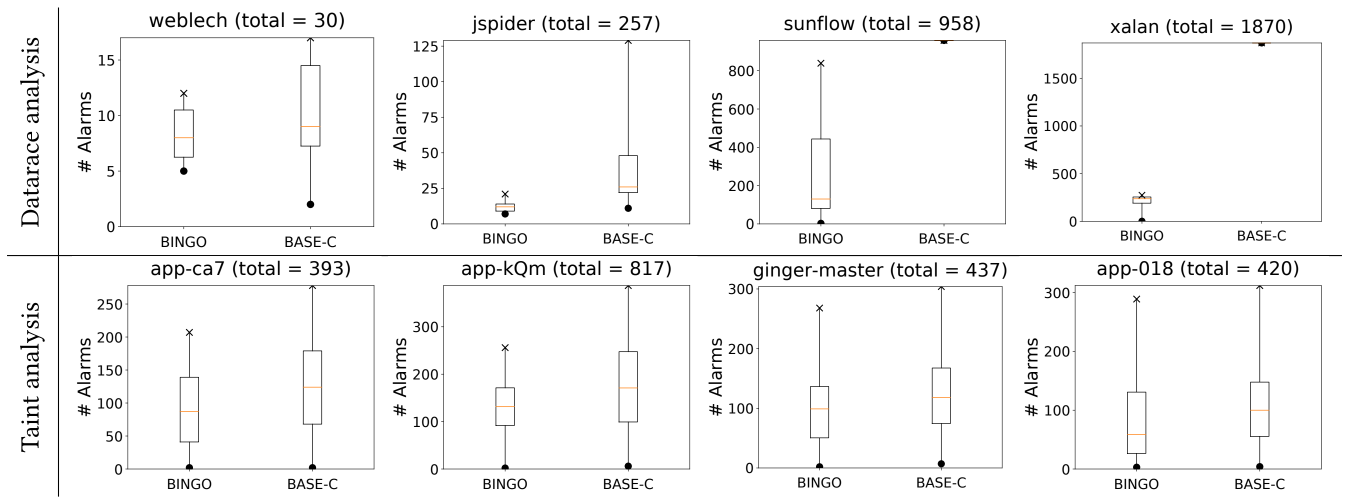

Figure 9. Comparing the runs of Bingo and Base-C.

Program	Bingo, optimized			Bingo, unoptimized			Base-C		
	#Tuples (K)	#Clauses (K)	Iter time (s)	#Tuples (K)	#Clauses (K)	Iter time (s)	#Vars (K)	#Clauses (K)	Iter time (s)
`app-324`	4	4	83	29	1,033	14,710	129	1,178	86
`noisy-sounds`	3	4	41	36	277	1,204	78	407	39
`app-ca7`	7	9	72	90	1,367	1,966	161	1,528	123
`app-kQm`	12	18	234	186	3,978	7,742	316	4,495	311
`tilt-mazes`	5	7	57	69	1,047	2,843	153	1,198	84
`andors-trail`	3	3	1	13	72	183	74	145	17
`ginger-master`	15	20	418	158	3,274	7,335	315	3,857	303
`app-018`	16	22	302	223	4,950	20,622	486	6,803	426

Table 8. Sizes of the Bayesian networks processed by Bingo, and of the MaxSAT problems processed by Base-C, for the eight Android apps, and their effect on iteration time.