RESEARCH-ARTICLE

# Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences

**TIANCHI LI**, Peking University, Beijing, China

**XIN ZHANG**, Peking University, Beijing, China

**Published:** 09 April 2025
**Accepted:** 18 February 2025
**Received:** 16 October 2024

# Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences

TIANCHI LI, Peking University, China
XIN ZHANG[*], Peking University, China

We propose a neural-symbolic style of program analysis that systematically incorporates informal information in a Datalog program analysis. The analysis is converted into a probabilistic analysis by attaching probabilities to its rules. And its output becomes a ranking of possible alarms based on their probabilities. We apply a neural network to judge how likely an analysis fact holds based on informal information such as variable names and String constants. This information is encoded as a soft evidence in the probabilistic analysis, which is a "noisy sensor" of the fact. With this information, the probabilistic analysis produces a better ranking of the alarms. We have demonstrated the effectiveness of our approach by improving a pointer analysis based on variable names on eight Java benchmarks, and a taint analysis that considers inter-component communication on eight Android applications. On average, our approach has improved the inversion count between true alarms and false alarms, mean rank of true alarms, and median rank of true alarms by 55.4%, 44.9%, and 58% on the pointer analysis, and 67.2%, 44.7%, and 37.6% on the taint analysis respectively. We also demonstrated the generality of our soft evidence mechanism by improving a taint analysis and an interval analysis for C programs using dynamic information from program executions.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: Program Analysis, Bayesian Inference, Soft Evidences, Neural Networks

**ACM Reference Format:**
Tianchi Li and Xin Zhang. 2025. Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 143 (April 2025), 28 pages. https://doi.org/10.1145/3720508

## 1 Introduction

Program analyses have made great strides in the past few decades. While there is a significant amount of effort in how to reason about behaviors of the programs relying on formal semantics, information that is informal but useful, is largely ignored in this process. Such information includes the naming of program elements (variables, procedures, classes, etc.), String constants, and comments. While this information does not give any formal guarantee, it can serve as good hints to estimate how likely a program fact holds. For example, variables that share the same name are more likely to alias to each other than variables that do not; a bug identified in a code fragment labeled with "TODO" is likely to be a true bug. It is highly desirable to incorporate such information in program analyses to improve their usability.

[*]Corresponding author.

Authors' Contact Information: Tianchi Li, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, litianchi@pku.edu.cn; Xin Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xin@pku.edu.cn.

ACM 2475-1421/2025/4-ART143
https://doi.org/10.1145/3720508

To achieve this, we need to address several challenges. First, how do we incorporate informal information without breaking the soundness of existing analyses? While this information is useful, it gives absolutely no soundness guarantees. Second, how do we automatically incorporate such information in existing program analyses? Ideally, we do not want to require significant manual effort to change existing analyses. Last but not least, how to extract informal information that is useful to the current analysis? Elements like variable names and comments contain a lot of informal information that is irrelevant to the analysis goal.

To address these challenges, we take a neural-symbolic approach that is expressed in the form of probabilistic logic programs together with neural networks. First, by attaching probabilities to logical analysis rules, it allows us to convert a conventional logic-based program analysis into a Bayesian model [30, 40]. By converting the analysis into a Bayesian model, the reports now come with probabilities, which can be used to rank them by how likely each is true. Initially, the probabilities for all reports depend on how they are derived. By considering informal information, the probabilities and the ranking will be updated. As a result, considering informal information will only update the ranking, but not break soundness. Second, by converting the analysis into a Bayesian model, we can easily extend the analysis to incorporate the informal information by extending the model. More concretely, we are going to use neural networks to estimate how likely a program fact derived by the analysis holds. Such information can be viewed as "noisy sensors" of the program facts. Following existing literature in Bayesian learning, we encode the neural network's output in the form of a kind of soft evidences [28], that are essentially noisy observations in Bayesian learning. Specifically, parameters of the soft evidences provide a way to quantify uncertainties in the informal information. This allows us to control the strength of each soft evidence in terms of how it affects the alarm ranking. There exist probabilistic programming languages that incorporate neural networks [16, 18], but none of them encode neural network outputs in the form of evidences, let alone soft evidences. In addition, our approach is the first to employ the soft evidence mechanism in Bayesian-style program analyses [4, 30, 40, 41].

We have implemented our idea in a framework called NESA for program analyses that are expressed in Datalog. We have instantiated NESA on a pointer analysis for Java programs where a neural network estimates how likely two variables alias with each other given their names. The alarms we consider are points-to facts. Besides, we also instantiated NESA on a taint analysis for Android applications by predicting how likely an ICC (inter-component communication) link is true given the related String constants. Our experiment shows that by considering information given by the neural network, our approach can improve the inversion count between true alarms and false alarms, mean rank of true alarms, median rank of true alarms by 55.4%, 44.9%, and 58% on the pointer analysis, and 67.2%, 44.7%, and 37.6% on the taint analysis respectively on average. In addition, to demonstrate the generality of our soft evidence mechanism, we have applied our approach to improve a taint analysis and an interval analysis for C programs by incorporating dynamic information from test runs.

In summary, our contributions are as follows:

(1) We propose a new paradigm in program analysis that combines formal information and informal information systematically.
(2) We propose a framework for the above paradigm by incorporating informal information in the form of soft evidences in a Bayesian program analysis. The informal information is extracted using a neural network.
(3) We demonstrated the effectiveness of our approach on a pointer analysis for Java programs and a taint analysis for Android programs.

```
1  Animal dolphin = new Dolphin();
2  Animal dog = new Dog();
3  Wrapper wrap = new Wrapper();
4  wrap.content = dolphin;
5  wrap.content = dog;
6  Animal dog1 = wrap.content;
7  describe(dog1);
8  // R1
9  Animal dog2 = (Dog)dog1;
10 interface Animal{}
11 class Dolphin implements Animal{}
12 class Dog implements Animal{}
13 class Wrapper{Animal content;}
14
15 void describe(Animal animal){
16 // R2
17     animal = (Dolphin)animal;
18     ...
19 }
```

Fig. 1. Example program.

```
pointsTo(V, H) :- allocation(V,H)
pointsTo(V, H) :- move(U, V), pointsTo(U, H)
fieldPointsTo(H1, F, H2) :- putField(V, F, U), pointsTo(V, H1), pointsTo(U, H2)
pointsTo(U, H2) :- getField(V, F, U), pointsTo(V, H1), fieldPointsTo(H1, F, H2)
unsafeDowncast(L) :- downcast(L, T1, V), pointsTo(V, H), typeOf(H, T2), !subType(
    T2, T1)
```

Fig. 2. A pointer analysis for checking downcasts.

## 2 Overview

We use an informal example that applies a pointer analysis to find unsafe type casts in a program to walk through our approach. Figure 1 shows the program which contains a safe type cast at line 9 and an unsafe cast at line 17. In particular, the method *describe* was implemented when only class *Dolphin* was present. At some point, class *Dog* was added but the method *describe* was not updated accordingly, which leads to an unsafe type cast. Figure 2 shows the pointer analysis, which is flow- and context-insensitive but field-sensitive. It is written in Datalog, a popular language to implement program analyses [40]. Briefly, each rule is an implication and the conditions are on the right-hand sides while the results are on the left-hand sides. When applying the analysis on the program, it reports two alarms *unsafeDowncast(9)* (denoted as R1) and *unsafeDowncast(17)* (denoted as R2).

The false alarm R1 is derived because the analysis is flow-insensitive. A conventional way to resolve such false alarms is to use finer abstractions, i.e., to make the analysis flow-sensitive. Here we take a different route, by taking hints from informal information such as variable names. We notice that the analysis derives *R*1 because it thinks *dog1* may point to the *Dolphin* object created at Line 1. This implies that variable *dog1* aliases with variable *dolphin*. To a human programmer, it seems not quite likely since the variable names suggest that they do not even have the same type. On the other hand, consider *R*2, variable *animal* pointing to the *Dog* object allocated at line 2 is more likely, as a "dog" is an "animal". The key challenge to incorporate such informal information is how to integrate it seamlessly into a rigorous formal analysis without breaking its correctness.

To address this challenge, we take a Bayesian view of the analysis, following previous works [17, 30]. By translating the analysis into a probabilistic one, we get a ranking of the alarms based on their beliefs. Adding the informal information will update such beliefs without adding or removing any alarm. Moreover, conditional inference based on the Bayes' theorem allows it easily to go from symptoms to roots, which automatically propagates the informal information to update the alarm beliefs. Compared to previous Bayesian approaches that integrate other external information, our

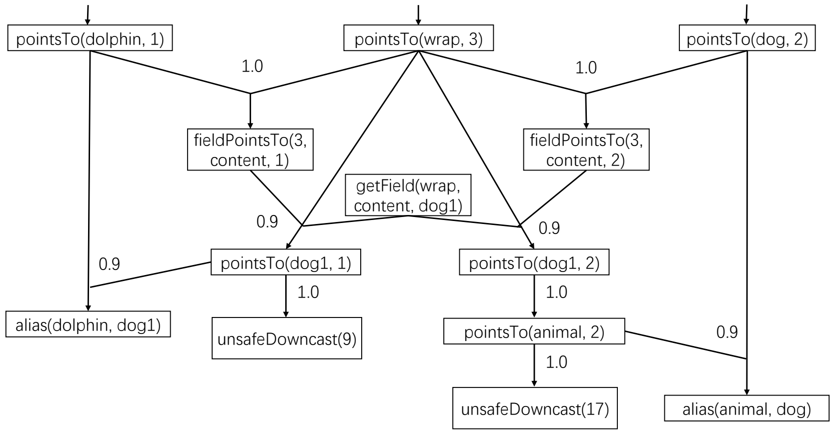


Fig. 3. Part of the Bayesian network derived from the analysis. Input tuples are ignored for space reasons.

key technical innovation is to employ the mechanism of *soft evidences*, which allows quantifying the noise in the informal information. We next describe the steps in detail.

*Adding New Rules to Relate Informal Information.* To enable conditional inference, we first need to find program facts that are related to the informal information. In this particular example, the corresponding program facts are pointer aliases. Since pointer aliases are not computed by the original analysis, we add the following rule:

```
alias(V,U) :- pointsTo(V, H), pointsTo(U, H)
```

*Converting into a Bayesian Analysis.* We next convert the analysis into a Bayesian one by attaching probabilities to each rule. Intuitively, the probability reflects how likely a rule holds in practice. The exact semantics follow that of Problog [7], a probabilistic logic programming language. How to set the exact weights is not the focus of the paper, one can get these weights by training from programs where the true alarms are known or relying on expert knowledge. In the empirical evaluation, we obtain the weights by training. Here, we notice that compared to the other rules, the fourth rule in the analysis that handles field gets is more likely to introduce false alarms. As a result, we assign a probability of 0.9 to this rule and probabilities of 1.0 to other rules. Besides, due to the abstraction of heap objects, the newly added rule that introduces alias facts is also approximate so we also assign a probability of 0.9 to it. Similarly, the weight of this rule can be obtained in a more systematic way like learning from labeled data. So the transformed analysis after adding probabilities is

```
pointsTo(V, H) :- allocation(V,H) 1.0
pointsTo(V, H) :- move(U, V), pointsTo(U, H) 1.0
fieldPointsTo(H1, F, H2) :- putField(V, F, U), pointsTo(V, H1), pointsTo(U, H2)
    1.0
pointsTo(U, H2) :- getField(V, F, U), pointsTo(V, H1), fieldPointsTo(H1, F, H2)
    0.9
unsafeDowncast(L) :- downcast(L, T1, V), pointsTo(V, H), typeOf(H, T2), !subType(
    T2, T1) 1.0
alias(V,U) :- pointsTo(V, H), pointsTo(U, H) 0.9
```

For the exact semantics of the probabilistic analysis, please refer to Section 3. In this example, since there is no circular dependency, its derivation can be interpreted as a Bayesian network (as shown in Figure 3). By calculating the marginal probabilities, we get a ranking of the two alarms: `R1` :

`unsafeDowncast(9) 0.9`, `R2: unsafeDowncast(17) 0.9`. The two reports have the same probability of 0.9 because there is only one approximate rule that is involved in deriving either of them. The ranking means that the Bayesian analysis does not give any additional information compared to the original analysis. Hence, a user can choose to inspect either first.

*Incorporating Informal Information.* We next incorporate informal information about `alias (dolphin, dog1)` and `alias(animal, dog)`. The problem here is: given a pair of variable names, how do we predict the likelihood that they alias with each other? The newly emerged community that studies deep learning for code and the natural language processing community have investigated a lot on this subject. For instance, we can use GraphCodeBert [9] to calculate the embedding of variable names. With the embedding, we can then design a simple neural network to calculate the likelihood. In this example, we construct a 2-layer fully-connected neural network. After fine-tuning, we can calculate that the likelihood that `alias(dolphin, dog1)` holds is 0.79, while the likelihood that `alias(animal, dog)` holds is 0.91. More generally, given an analysis fact tuple $r(d_1, .., d_n)$, we assume there exists an embedding for each constant $d_i, i \in [1, n]$, and one can train a machine learning model to estimate how likely a tuple from relation $r$ holds based on the embedding using labeled data.

Our next challenge is how to incorporate such likelihood in the Bayesian analysis. A naive approach is to add them as conventional (hard) evidences. More concretely, we can set a threshold, say 0.5, and any likelihood value above it is treated as a positive evidence while any value below it is treated as a negative evidence. However, picking this threshold is tricky. In fact, in this example, we will get two positive evidences on the alias tuples, which leads to no change in the ranking. Moreover, these likelihoods contain different degrees of uncertainties, reflecting the imprecise nature of many machine learning methods. It is a matter of *more likely vs. less likely* rather than *yes vs. no.* Simply converting these likelihoods into binary evidences will lose such information.

To resolve this challenge, we leverage the idea of *soft evidences* [5]. Soft evidences offer a mechanism to update beliefs in a Bayesian system based on observations from "noisy sensors". The particular kind we apply is called "virtual evidences". In particular, for each *alias* node, we add a new node *alias'* that corresponds to whether the neural component thinks the corresponding fact holds based on the variable names. To make it work, we need to obtain $P(alias' \mid alias)$ and $P(alias' \mid \neg alias)$. We can get these conditional probabilities using the aforementioned likelihood calculated by the neural network. Intuitively, we always give a positive observation on *alias'*. The higher the likelihood is, the higher $P(alias' \mid alias)$ and $P(\neg alias' \mid \neg alias)$ are, and therefore the lower $P(\neg alias' \mid alias)$ and $P(alias' \mid \neg alias)$ are. Concretely, we directly leverage the probabilities from the output of the neural network:

$$P(alias' \mid alias) = P(\neg alias' \mid \neg alias) = NN(v_1, v_2)$$

In the above formula, $v_1$ and $v_2$ are variable names. NN is the neural network whose output is the likelihood that $v_1$ and $v_2$ alias with each other. Following the above recipe, the updated ranking is `R2: unsafeDowncast(17) 0.98, R1: unsafeDowncast(9) 0.95`. Now by considering informal information, the true alarm comes before the false alarm.

## 3 Preliminaries

We introduce necessary notations before introducing our approach. We use a probabilistic logic language [33] to specify our analysis, which extends Datalog. Our syntax and semantics largely follow those of Problog [7]. Figure 4 shows the detail.

$$\begin{array}{rlrl}
(\mathit{Probabilistic\ Program}) & C_p ::= (\bar{c}_p, \bar{e}) & (\mathit{Probabilistic\ Rule}) & c_p ::= n\ c\ p \\
(\mathit{Program}) & C ::= \bar{c} & (\mathit{Rule}) & c ::= l \,\text{:-}\, \bar{l} \\
(\mathit{Literal}) & l ::= r(\bar{a}) & (\mathit{Argument}) & a ::= d \mid v \\
(\mathit{Evidence}) & e ::= \mathit{observe}(t) \mid \mathit{observe}(!t) & (\mathit{Tuple}) & t ::= r(\bar{d})
\end{array}$$

(a) Syntax of Probabilistic Datalog

$$\begin{array}{rlrl}
(\mathit{Rule\ Name}) & n \in \mathbb{N} & (\mathit{Rule\ Probability}) & p \in [0,1] \\
(\mathit{Relation\ Name}) & r \in \mathbb{R} & (\mathit{Constant}) & d \in \mathbb{D} \\
(\mathit{Variable}) & v \in \mathbb{V} & (\mathit{Substitution}) & \sigma \in \mathbb{V} \mapsto \mathbb{D} \\
(\mathit{Ground\ Probabilistic\ Rule}) & pgc ::= n\ gc\ p & (\mathit{Ground\ Rule}) & gc ::= t : -\bar{t}
\end{array}$$

(b) Auxiliary definitions

$$\begin{array}{rl}
P_{[\![(\bar{c}_p,\bar{e})]\!]_\omega}(T) & = \sum_{PGC \subseteq U_{\bar{c}_p} \wedge [\![PGC]\!]_D = T} P_{[\![(\bar{c}_p,\bar{e})]\!]_\omega}(PGC) \\
P_{[\![(\bar{c}_p,\bar{e})]\!]_\omega}(PGC) & = \begin{cases} \frac{P_{[\![\bar{c}_p]\!]_\omega}(PGC)}{\sum_{PGC' \subseteq U_{\bar{c}_p} \wedge PGC' \models_D \bar{e}} P_{[\![\bar{c}_p]\!]_\omega}(PGC')} & \text{if } PGC \models_D \bar{e} \\ 0 & \text{otherwise} \end{cases} \text{, where } PGC \subseteq U_{\bar{c}_p} \\
P_{[\![\bar{c}_p]\!]_\omega}(PGC) & = \prod_{(n\ gc\ p) \in PGC \cap U_{\bar{c}_p}} p \times \prod_{(n\ gc\ p) \in U_{\bar{c}_p} \setminus PGC} (1-p) \text{, where } PGC \subseteq U_{\bar{c}_p} \\
U_{\bar{c}_p} & = \{pgc \mid \exists \sigma, c_p \in \bar{c}_p . pgc = \sigma(c_p)\} \qquad p\bar{g}c \models_D \bar{e} \Leftrightarrow \bigwedge_{e \in \bar{e}} pgc \models_D e \\
p\bar{g}c \models_D \mathit{observe}(t) & \Leftrightarrow t \in [\![p\bar{g}c]\!]_D \qquad p\bar{g}c \models_D \mathit{observe}(!t) \Leftrightarrow t \notin [\![p\bar{g}c]\!]_D \\
[\![p\bar{g}c]\!]_D & = \text{lfp } S_{\{gc \mid n\ gc\ p \in p\bar{g}c\}} \text{, where } S_{\bar{g}c}(T) = T \cup \{t \mid \exists t \,\text{:-}\, \bar{t} \in \bar{g}c \wedge \bar{t} \subseteq T\}
\end{array}$$

(c) Semantics of Probabilistic Datalog

Fig. 4. Syntax and semantics of the probabilistic Datalog.

A probabilistic program is a set of probabilistic rules and a set of evidences[1]. A probabilistic rule is a triple comprising a unique rule name, a conventional logical rule, and a probability. The logical rules are in the standard Datalog form: a head literal (the result), followed by a list of body literals (the condition). A literal is a relation name that is followed by a list of variables or constants. We refer to a literal with only constants a *tuple*. An evidence is a tuple or a negated tuple.

A probabilistic program defines a distribution of Datalog programs, which in turn defines a distribution of Datalog outputs (i.e., sets of tuples). We can define the distribution of Datalog programs using a sampling process. First, each probabilistic rule is converted to a set of ground probabilistic rules by replacing all variables with all possible values. In Figure 4, we denote a possible replacement using $\sigma$. For simplicity, we have slightly abused this notation so it also applies to a probabilistic rule where all variables are replaced according to $\sigma$. We use $U_{\bar{c}_p}$ to denote all the ground probabilistic rules that are obtained from a given probabilistic program. Note each ground rule inherits the probability from the original rule. Second, we sample a subset of the ground rules based on their probabilities. The probability of a sampled subset $PGC$ is denoted by $P_{[\![\bar{c}_p]\!]_\omega}(PGC)$. However, we also need to consider evidences, which is standard in terms of computing conditional probabilities. Briefly, the probability of a subset program is 0 if its deterministic counterpart can (not) derive tuple $t$ for any $\mathit{observe}(!t)$ ($\mathit{observe}(t)$); the probabilities of other subset programs

[1]We use $\bar{a}$ to denote a set of $a$'s.

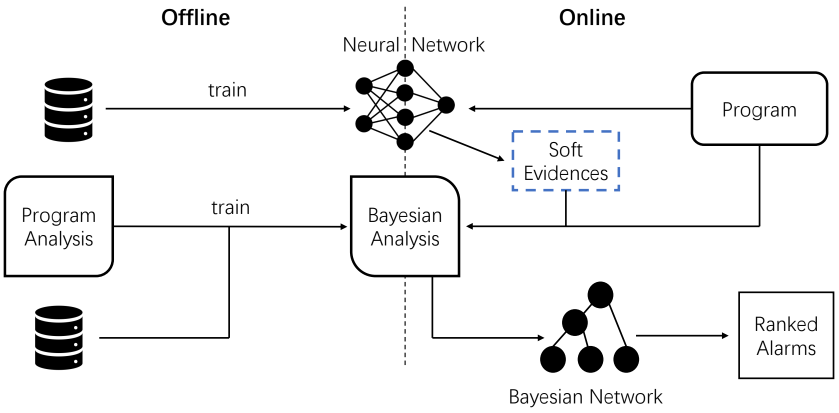


Fig. 5. Overall workflow of our approach.

that satisfy evidences need to be re-normalized. We use $[\![ \bar{pgc} ]\!]_D$ to denote evaluating a set of probabilistic rules deterministically, that is, evaluating its Datalog counterpart after removing the probabilities. For a non-zero probability subset program, its probability is then normalized based on all the subset programs which have non-zero probabilities. Finally, the probability of a given set of tuples $T$ is computed by summing up the probabilities of all the subset programs that can derive them. We use $P_{[\![(\bar{c}_p,\bar{e})]\!]_\omega}(T)$ to denote this probability. With it, we can compute the marginal probability of a given tuple $t$:

$$P_{[\![(\bar{c}_p,\bar{e})]\!]_\omega}(t) = \sum_{t \in T} P_{[\![(\bar{c}_p,\bar{e})]\!]_\omega}(T).$$

It is clear that as long as the positive evidences only consist of the facts derivable from the Datalog counterpart, any fact that cannot be derived by the Datalog counterpart must have a probability of 0. This ensures the soundness of our approach.

## 4 Overall Framework

Figure 5 shows the workflow of our approach, which is divided into an offline phase and an online phase. In the offline phase, a conventional analysis is converted into a Bayesian analysis by attaching probabilities to its rules. These probabilities can be trained from programs whose true alarms are known [7]. A neural network is also trained to produce noisy observations on program facts. We will talk about the format of such observations and general guidelines to compose such a network later. In the online phase, given a program to analyze, the neural network produces noisy observations on program facts which are relevant to the analysis. Then the Bayesian analysis is converted into a Bayesian network along with the noisy observations. Finally, our framework performs marginal inference on the Bayesian network to produce a ranking of alarms.

## 5 Online Inference With Soft Evidence

In this section, we explain how our framework performs alarm ranking with soft evidences in detail. Algorithm 1 outlines the process. The input to the algorithm includes a Bayesian program analysis $pa$, a program $T$ to analyze, a relation $r(\bar{v})$ whose tuples will be observed with soft evidences, and a neural network $NN$ to give a soft evidence to each program fact in the relation $r(\bar{v})$ which indicates how likely it holds. Both the Bayesian analysis and the neural network are obtained through training in the offline phase, which we will describe in Section 6. Suppose the logical

**Algorithm 1** Online Inference.

```
Require: A Bayesian program analysis $pa$, a program $T$, a relation $r(\bar{v})$, a neural network $NN$.
Ensure: A ranking of analysis alarms.
1: softEvi := $\emptyset$
2: analysisResult := runLogicalAnalysis($pa$, $T$)
3: for $t \in$ analysisResult do
4:     if $t$ is in $r(\bar{v})$ then
5:         $sv := NN(t)$
6:         softEvi := softEvi $\cup\{sv\}$
7: bayesNet := compile($pa$, softEvi)
8: alarmProbMap = marginal(bayesNet)
9: return rankAlarmsByMarginals(alarmProbMap)
```

counterpart of the analysis is sound, then we only care about facts in $r(\bar{v})$ that can be derived by it. This is because any fact that cannot be derived has a probability of 0 to hold. We obtain such probable facts by running the logical analysis (line 2). Then the algorithm iterates the tuples in the analysis result and applies the neural network to produce soft evidence for each tuple in relation $r(\bar{v})$ (line 3-8). Intuitively, the soft evidence indicates how likely the program fact holds. The Bayesian analysis is then compiled into a Bayesian network along with the soft evidences (line 9). Finally, the algorithm ranks the analysis alarms based on the marginal probabilities of the alarms that are computed using the Bayesian network.

In the rest of the section, we first describe what the soft evidences are and how they are implemented, and then describe how the ranking problem is compiled into an inference problem on a Bayesian network.

## 5.1 Definitions of Soft Evidences

Unlike conventional hard evidences, soft evidences do not assert a fact absolutely holds or does not hold, but update the beliefs about the fact instead. There are multiple kinds of soft evidences, and the specific kind we apply is called "virtual evidences". We follow the "nothing else considered" method and introduce the definitions of soft evidences formally [5].

Definition 1 (Odds of an Event). *The odds of an event is the probability that it holds divided by the probability that it does not hold. That is*

$$O(\beta) = \frac{P(\beta)}{P(\neg\beta)}.$$

If $O(\beta) = 1$, it means we believe event $\beta$ holds and $\beta$ does not hold equally; if $O(\beta) = 10$, we believe $\beta$ holds 10 times more than we do not. Soft evidences update the odds of given events.

Definition 2 (Soft Evidences). *A soft evidence has the form of*

$$observe(\beta)\ k,$$

*where $\beta$ is an event and $k$ is a non-negative real number. After observing the soft evidence, the odds of $\beta$ will be increased by $k$ times, that is*

$$\frac{O(\beta \mid observe(\beta)\ k)}{O(\beta)} = k.$$

In the case of a Bayesian program analysis that is specified in probabilistic Datalog, an event $\beta$ is denoted as a tuple $t$ which represents a program fact. The value $k$ is known as the *Bayes factor.*

To obtain such soft evidences, we apply a neural network to estimate how likely a program fact holds. However, instead of producing the Bayes factors, our neural network directly produces key parameters that are used in the soft evidence encoding, which is implemented as conditional probabilities with conventional evidences. The next subsection introduces the encoding in detail.

When the original analysis is sound, it is easy to see that any fact in the observed relation that is not derived has a probability of 0. As a result, we only apply the neural network to estimate an analysis fact that is derived by the original analysis.

## 5.2 Encoding Soft Evidences using Conditional Probabilities

We now introduce how to perform inferences with soft evidences. Concretely, our approach first encodes soft evidences as conditional probabilities with conventional evidences. This follows the idea of interpreting soft evidences as "noisy sensors" in probability calculus [5]. Then our approach computes marginal probabilities of alarms, which are used to rank alarms.

A soft evidence can be seen as a "noisy sensor" on the event it is observed on. In other words, when the evidence is observed, the corresponding event has a probability to be true or false. More concretely, for a program fact $t$ that the soft evidence is observed on, we add another event $t'$ which serves as a noisy sensor of $t$. For example, in the pointer analysis example in Section 2, *alias'(dolphin, dog1)* and *alias'(animal, dog)* are added and linked to the corresponding alias tuples. The noisy sensor has a true positive rate and a true negative rate as its parameters which are denoted by $P(t' \mid t)$ and $P(\neg t' \mid \neg t)$ respectively. The Bayes factor $k$ determines them. In other words, a soft evidence $observe(t)\ k$ is converted into

$$observe(t') \text{ with } P(t' \mid t) = f(k), P(\neg t' \mid \neg t) = g(k).$$

Note we have assumed that a soft evidence is always positive, because a negative soft evidence can be converted into a positive one by changing the true positive rate and the true negative rate. In other words, a negative soft evidence $observe(\neg t)\ k$ with $t'$ as its noisy sensor can be converted into $observe(t'')$ where $t'' = \neg t'$.

We now explain how to deduce $P(t' \mid t)$ and $P(\neg t' \mid \neg t)$ from the Bayes factor $k$. We have

$$\begin{aligned} O(t \mid observe(t)\ k) = O(t \mid t') &= \frac{P(t \mid t')}{P(\neg t \mid t')} \\ &= \frac{P(t' \mid t) \times P(t)/P(t')}{P(t' \mid \neg t) \times P(\neg t)/P(t')} && \text{(by Bayes' Theorem)} \\ &= \frac{P(t' \mid t)}{P(t' \mid \neg t)} \times \frac{P(t)}{P(\neg(t)} = \frac{P(t' \mid t)}{1 - P(\neg t' \mid \neg t)} \times O(t). \end{aligned}$$

From the above equation, we can further derive

$$k = \frac{O(t \mid observe(t)\ k)}{O(t)} = \frac{P(t' \mid t)}{1 - P(\neg t' \mid \neg t)}.$$

The above equation shows that we can control the Bayes factor in the soft evidence by changing the true positive rate $P(t' \mid t)$ and true negative rate $P(\neg t' \mid \neg t)$ of the noisy sensor $t'$. Moreover, when the true negative rate goes to 1 and the true positive rate is nonzero, $k$ approaches infinity. This will make the soft evidence a conventional hard evidence.

Conversely, once $k$ is given, we can construct the noisy sensor and the corresponding conditional probabilities accordingly. Note, the above deduction shows that the absolute values of the conditional probabilities do not matter. One can add the following rules to the probabilistic analysis:

$$t' :\!- \, t \; P(t' \mid t) \text{ and } t' :\!- \, \neg t \; (1 - P(\neg t' \mid \neg t)).^{2}$$

Instead of computing the Bayes factor, our neural networks are directly trained to produce the conditional probabilities. We will talk about the details in Section 6. To perform inference, the soft evidences are compiled as a part of a Bayesian network along with the original Bayesian program analysis, which we will introduce next.

## 5.3 Compiling into a Bayesian Network

We now introduce how to use the probabilistic program analysis with soft evidences to rank alarms. Concretely, we compute the marginal probabilities of each alarm and rank them based on these probabilities - alarms with higher probabilities are ranked higher. Given the augmented analysis can be converted into a probabilistic Datalog program, we can perform marginal inference with engines of existing probabilistic logical programming languages such as Problog. However, we have tried Problog's inference engine and it failed to terminate on the datasets in our experiment. On one hand, the algorithms in Problog are either exact or sampling-based [32] - they are geared towards accuracy. On the other hand, the marginal inference problem is a counting problem and the numbers of relevant tuples and ground probabilistic rules per program in our pointer analysis experiment are from 896 thousand to 3.2 million and 999 thousand to 19.4 million respectively [3]. As a result, for scalability, we follow the approach of Bingo [30], and perform an approximate inference by converting the Bayesian analysis into a Bayesian network.

First, we only need to consider tuples that can be derived by the deterministic version of the analysis and their related ground rules. This is because when there is no negation in the rule bodies, the tuples that a set of ground (deterministic) rules can derive are monotone. In other words, removing any ground rule from the set will not increase the derived tuples. As a result, any tuple that cannot be derived by the full set of ground rules in the analysis has a probability of 0 to hold.

Second, there can be loops in terms of analysis derivations while Bayesian networks do not permit loops. One approach is to convert the analysis derivations into a Boolean formula and then convert the formula into a factor graph. Factor graphs allow loops and mainstream Bayesian network inference implementations typically convert Bayesian networks into factor graphs to perform the actual inference. However, it is nontrivial to encode the least fixed semantics in a Boolean formula when loops are present. Existing approaches will blow up the graph size significantly [10, 20]. As a result, we apply Bingo's approach to remove cycles in the analysis, which will lead to approximations in the results. Briefly, when there are multiple paths to derive a tuple, only the shortest paths will be kept. For details, please refer to the paper by Raghothaman et al. [30].

Once cycles are removed, the probabilistic analysis is converted into a Bayesian network. We create a Boolean random variable for each tuple that is derived by the deterministic analysis. Given a tuple $t$, suppose there are $n$ ground probabilistic rules that can derive it. We create $n$ random variables $t_1, t_2, ..., t_n$, which represent different rules to derive $t$. We add an edge from them to $t$ with the following conditional probabilities:

$$P(t \mid \bigvee_{i \in [1,n]} t_i) = 1 \text{ and } P(t \mid \bigwedge_{i \in [1,n]} \neg t_i) = 0.$$

[2]The syntax of our probabilistic Datalog rule does not allow negations in the rule bodies. However, the negations here are stratified. So following semantics of stratified negations in Datalog, the semantics is still well-defined.
[3]The relevant tuples and ground rules are the ones that are involved in the least fixed-point of the Datalog counterpart of the probabilistic program.

For the $i$th rule, let $t_i^1, ..., t_i^m$ $p$ be the tuples in its body, we add an edge from all the body tuples to $t_i$ with the following conditional probabilities:

$$P(t_i \mid \bigwedge_{j \in [1,m]} t_i^j) = p \text{ and } P(t_i \mid \bigvee_{j \in [1,m]} \neg t_i^j) = 0.$$

For example, suppose we have two ground probabilistic rules $A :\!- B, C$ 0.8 and $A :\!- D, E$ 0.7. We create random variables $A, A_1, A_2, B, C, D, E$, and add edges with conditional probabilities $P(A \mid A_1 \vee A_2) = 1, P(A \mid \neg A_1 \wedge \neg A_2) = 0, P(A_1 \mid B \wedge C) = 0.8, P(A_1 \mid \neg B \vee \neg C) = 0, P(A_2 \mid D \wedge E) = 0.7, P(A_2 \mid \neg D \vee \neg E) = 0$. After constructing the Bayesian network, we add the soft evidences' conditional probability encoding to the network by the method described in the previous subsection.

Finally, we compute marginal inference on the Bayesian network by converting it into a factor graph and performing loopy belief propagation on the graph. One way to measure the quality of the result is the number of inversions between true alarms and false alarms. If we rank the alarms based on the marginal probabilities, the number of inversions is the number of pairs where a false alarm is ranked before a true alarm. Similar to the previous work [30], we can prove that ranking by the marginal probabilities has the minimum expected inversion count according to the joint distribution defined by the Bayesian program analysis with soft evidences. The proof of the following theorem is shown in Appendix A.

THEOREM 1. *Let $P$ be the joint distribution of program facts defined by the Bayesian program analysis with soft evidences, ranking alarms by the marginal probabilities has the least expected inversion count according to $P$.*

## 6 Offline Learning

The offline phase of our framework turns a conventional logical analysis into a Bayesian one and trains a neural network. We first briefly describe the former, and then give a general guideline to train a neural network for our purpose. In particular, we will describe how to encode the output of the neural network for a given program fact as the corresponding soft evidence.

### 6.1 Generating a Bayesian Program Analysis

One purpose of our offline phase is to convert a conventional program analysis expressed in Datalog into a Bayesian one expressed in our probabilistic Datalog. This mainly involves attaching a probability to each analysis rule indicating how likely it holds. One can either specify it manually or train it from labeled data [7]. We follow the latter in our evaluation and assume there are a set of programs whose true alarms are known. We apply the expectation-maximization algorithm [23].

Sometimes, it is hard to directly provide additional information on analysis results based on informal information but easy to provide information on facts that are further derived from analysis results. In this case, additional rules need to be added to the analysis to derive these facts. Consider the pointer analysis in Section 2. It is relatively easy to predict whether two variables alias from each other based on their names. On the other hand, it is harder to predict whether a variable points to a certain abstract object (denoted by a *pointsTo* tuple). As a result, a rule *alias(V, U)* :- *pointsTo(V, H), pointsTo(U, H)* is added to the analysis to produce tuples in the *alias* relation. Given program analyses typically approximate, these rules will very likely also approximate. We expect the choices of approximations in these rules are consistent with those of the original analysis. In other words, if the original analysis over-approximates, the new rules should also over-approximate. Similarly, when the analysis is later made probabilistic, these newly added rules will also become probabilistic (via training or manual specification). In the pointer analysis example, the added rule over-approximates as an abstract object $H$ over-approximates a set of concrete objects.

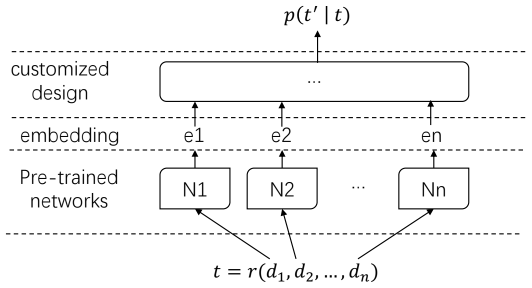


Fig. 6. Architecture of our neural network. The pre-trained networks can be the same one.

## 6.2 Training a Neural Network Output to Produce Soft Evidences

In this section, we describe general guidelines to design a neural network for our purpose. In particular, we first describe the output of the network and then the input features.

The input to our training algorithm is a set of programs whose facts in the relation $r(\bar{v})$ (the relation on which soft evidences operate) are labeled to indicate whether they hold. We only care about the facts that can be derived by the logical analysis. Recall that, given a tuple $t$ in $r(\bar{v})$, our approach creates a noisy sensor $t'$, which is parameterized by $P(t' \mid t)$ and $P(\neg t' \mid \neg t)$. Our goal is to train the neural network to produce these two conditional probabilities for $t$ such that the probability of the noisy sensor $t'$ holding is maximized given the truth of $t$. This can be achieved by using the maximum likelihood estimation method, which is to maximize the following probability:

$$\prod_{t \in T} (label(t) \times P(t' \mid t) + (1 - label(t))(1 - P(\neg t' \mid \neg t))).$$

Here $T$ is the set of program facts in the training data, and $label$ indicates whether a fact holds according to the oracle. It is easy to see that the above probability is maximized if $P(t' \mid t) = 1$ when $label(t) = 1$ and $P(\neg t' \mid \neg t) = 0$ when $label(t) = 0$.

Now for a given pair $(t, t')$ in the training data where $t'$ is the noisy sensor of $t$, we know whether $t$ holds and $t'$ always holds (according to our encoding in Section 5.2). Learning functions producing $P(t' \mid t)$ and $P(\neg t' \mid \neg t)$ separately would be under-constrained. Since $t'$ always holds, the former only has positive points and the latter only has negative points. This is in line with that different pairs of true positive rate and true negative rate may produce the same Bayes factor (recall that $k = P(t' \mid t)/(1 - P(\neg t' \mid \neg t))$). To address this issue, we assume that $P(t' \mid t) = P(\neg t' \mid \neg t)$. As a result, our neural networks are trained to produce one kind of conditional probabilities. Recall that the optimization objective (i.e., the probability above) will prefer to maximize $P(t' \mid t)$ when $t$ holds and minimize $P(\neg t' \mid \neg t)$ when $t$ does not hold. After making $P(t' \mid t) = P(\neg t' \mid \neg t)$, $P(t' \mid t)$ in fact estimates how likely $t$ holds.

The input to our neural network is a program fact and its related program. How to design the features and the network architecture is a dedicated issue and affects the performance of our approach. We provide general guidelines on these two issues. Recall that a tuple $r(d_1, ..., d_n)$ is a relation name followed by a list of constants which are usually program elements such as statements. To produce the conditional probabilities, one would expect to extract features related to these program elements. This may sound nontrivial. Fortunately, how to extract embedding from programs is an active area [1, 2, 9, 12, 34] and we can leverage their progress. As a result, we would suggest to use these pre-trained networks to form the lower layers of the network. For each constant in the tuple, an embedding is produced. Then the rest of the network is built atop them. That can be relatively easy, since a naive fully-connected network is often sufficient for the task.

We expect most of the complexity of the network to go into the lower embedding layers which are handled by existing pre-trained networks. Figure 6 shows our overall design of the neural network.

The network in Section 2 can be seen as an instance of this design. The tuples of concern are in relation *alias* where the constants are variables. We use GraphCodeBert [9] to extract embeddings from their names, which serve as the lower layers of our network. Then the fully-connected network can be viewed as the upper layer. Note these pre-trained models such as GraphCodeBert can be fine-tuned together in the overall training process.

## 7 Empirical Evaluation

In this section, we evaluate the effectiveness of our approach by applying it to a pointer analysis on a suite of 8 Java programs and a taint analysis on 8 Android applications, considering two different kinds of informal information respectively.

### 7.1 Evaluation Setup

We implemented our framework NESA atop Bingo [30], which is a framework that can convert a Datalog analysis into a Bayesian network. We use Chord [24] as the Java analysis frontend, and LibDAI version 0.3.2 [22] as the Bayesian network inference backend. All experiments were done using Oracle HotSpot JVM 1.6 on a Linux machine with 256GB memory and 2.6GHz processors. Next, we first introduce the analysis and the benchmarks, and then describe how each component of NESA is configured.

*7.1.1 Analysis and Oracle.* We evaluated NESA on two instance analyses: a pointer analysis for Java programs and a taint analysis for Android applications.

*Pointer analysis.* The pointer analysis we consider is 0-CFA, which is a context- and flow-insensitive pointer analysis on Java programs. The alarms we consider are points-to facts.

*Taint analysis.* The taint analysis [6] is applied to expose privacy leaks in Android applications. Specifically, sensitive variables in the analyzed program are labelled with *source* and *sink* annotations. The goal of the analysis is to detect data flows from *source* to *sink*, which represent possible privacy leaks. It relies on the pointer analysis we mentioned above to get call-graph and aliasing information.

The focus of studying this analysis is to use informal information to mitigate the imprecision incurred by inter-component communications in Android applications. Different from Java programs, Android applications are made up of basic units called components. Components may communicate with each other, which can then cause inter-component privacy leaks. This kind of communication (called ICC) consists of two main artifacts: *Intent* and *Intent Filter*. Specifically, one component can start another component by calling a specific ICC method with an *Intent* instance as its parameter. And if the *Intent*'s fields (e.g., `action`, `category`) can match those of another component's *Intent Filter*, then the communication is established. As shown in a previous work [13], ICC analysis is an important part of detecting inter-component leaks. However, existing ICC analysis tools like IC3 [26] can introduce imprecision in *Intent* fields, which then leads to false positives of ICC links. Previous research has shown that one can use a neural network to predict whether an ICC link holds [42]. So we focus on predicting those ICC links using a neural network in our evaluation.

Unfortunately, the analysis implementation we consider comes from Bingo's artifact [30], which cannot support inter-component analysis. As a solution, we simulate ICCs by substituting a subset of data flows via method invocations with ICC links, which can be true or spurious. The distribution of true/false ICC links and the pattern of ICCs are taken from the paper by Octeau et al. [25]. Those data flows are represented by the "transferRefRef" relation in the original analysis, and we change part of them into `ICC(I, F)`, standing for an intent and an intent filter from an ICC link.

```
public class Outflow extends Activity {
    protected void onCreate() {
        String source1 = ... //source1
        String source2 = ... //source2
        //action = "android.appwidget.action"
        String actionStr = Config.actionMap.get("Outflow");
        //i1.action = "android.appwidget.action"
        Intent i1 = new Intent(actionStr);
        //i2.action = "android.appwidget.action.APPWIDGET_UPDATE"
        Intent i2 = new Intent(actionStr + ".APPWIDGET_UPDATE");
        i1.putExtra("DeviceId", source1);
        i2.putExtra("DeviceId", source2);
        startActivity(i1); //Failed ICC link
        startActivity(i2); //Successful ICC link
    } }
```

Fig. 7. Example program to explain the informal information used in the taint analysis.

*Oracle.* To label which alarms are true, we run more precise analyses. For the pointer analysis, we run a context-sensitive 2-object sensitivity analysis [21]. For the taint analysis, we run a context-sensitive 3-object sensitivity pointer analysis to get call-graph and aliasing information. Besides, the ground truths of our newly inserted ICC links are also taken into concern to label oracle alarms.

### 7.1.2 *Benchmarks.*

Table 1 shows the characteristics of our selected benchmarks. The "#Clauses" column shows the number of ground rules in the least fixed point of the analysis on each benchmark. It roughly corresponds to the number of edges in the corresponding Bayesian network. This demonstrates the scales of the inference problems we consider. The "#Alarms" columns show the number of true, false, and all alarms.

### 7.1.3 *Extracting Informal Information and the Neural Network.*

We consider two different kinds of informal information for the two instance analyses, which are variable names and String constants related to inter-component communication (ICC), respectively. We will discuss them in detail next.

*Pointer analysis.* Similar to the example in Section 2, we use a neural network to score how likely an alias pair is true based on variable names. We add a rule `alias(V, U) :- pointsTo(V, H), pointsTo(U, H)` to the analysis to compute alias pairs. Further, we only consider aliases both of whose variables are in the application code. One issue is that variables in the analysis domain is at the intermediate language level. Not all variables can be mapped back to variables in the source code. As a result, we only consider alias pairs both of whose variables can. The "#Evidences" column in Table 1 shows the number of these alias pairs. The neural network we use is GraphCodeBert [9], which is a pre-trained model to produce embedding for program elements. We design a naive 2-layer fully-connected neural network for alias prediction. This 2-layer neural network consists of an input layer, a 128-unit hidden layer with Leaky ReLU activation, and a Sigmoid output layer to get the probability that an alias pair is true. There might be more complex and advanced code models, but GraphCodeBert is sufficient to demonstrate the effectiveness of our approach. Using better neural networks will only improve our results.

*Taint analysis.* The informal information we consider for the taint analysis is String constants related to inter-component communication (ICC) as mentioned above. Since ICC matching is essentially a String matching problem between the fields of *Intent* and *Intent Filter*, we can use

Table 1. Benchmark characteristics

(a) Benchmark characteristics of pointer analysis. (k = thousand)

| | Description | #Classes | | #Methods | | Bytecode(KB) | |
|---|---|---|---|---|---|---|---|
| | | app | total | app | total | app | total |
| montecarlo | financial simulator | 18 | 115 | 114 | 444 | 5.2 | 22.7 |
| moldyn | molecular dynamics simulator | 10 | 73 | 40 | 230 | 6.2 | 18 |
| weblech | website download/mirror tool | 56 | 579 | 302 | 3,344 | 19.3 | 208.4 |
| toba-s | Java bytecode to C compiler | 25 | 159 | 149 | 747 | 32 | 55.9 |
| hedc | web crawler from ETH | 44 | 357 | 230 | 2,154 | 15.8 | 140.2 |
| jspider | web spider engine | 113 | 391 | 422 | 1,572 | 17.7 | 74.6 |
| javasrc-p | create HTML pages | 49 | 136 | 461 | 791 | 43 | 59.7 |
| ftp | Apache FTP server | 119 | 496 | 608 | 2,751 | 36.5 | 142.9 |

| | #Clauses | #Alarms | | | #Evidences |
|---|---|---|---|---|---|
| | | true | false | total | |
| montecarlo | 1,166k | 615 | 3,601 | 4,216 | 2,216 |
| moldyn | 999k | 640 | 1,760 | 2,400 | 256 |
| weblech | 9,020k | 2,562 | 23,451 | 26,013 | 2,604 |
| toba-s | 5,590k | 4,579 | 233,849 | 238,428 | 10,128 |
| hedc | 5,460k | 2,425 | 4,660 | 7,085 | 14,214 |
| jspider | 3,082k | 3,336 | 19,990 | 23,326 | 19,852 |
| javasrc-p | 19,450k | 34,089 | 43,639 | 77,728 | 84,968 |
| ftp | 14,709k | 6312 | 68,717 | 75,029 | 120,214 |

(b) Benchmark characteristics of taint analysis. (k = thousand)

| | Description | #Classes | | #Methods | |
|---|---|---|---|---|---|
| | | app | total | app | total |
| app-324 | Unendorsed Adobe Flash player | 81 | 1,788 | 167 | 6,518 |
| noisy-sounds | Music player | 119 | 1,418 | 500 | 4,323 |
| app-ca7 | Simulation game | 142 | 1,470 | 889 | 4,928 |
| app-kQm | Puzzle game | 105 | 1,332 | 517 | 4,114 |
| tilt-mazes | Game packaging the Mobishooter malware | 547 | 2,462 | 2,815 | 7,034 |
| andors-trail | RPG game infected with malware | 339 | 1,623 | 1,523 | 5,016 |
| ginger-master | Image processing tool | 159 | 1,474 | 738 | 4,500 |
| app-018 | Arcade game | 275 | 1,840 | 1,389 | 5,397 |

| | Bytecode(KLOC) | | #Clauses | #Alarms | | | #Evidences |
|---|---|---|---|---|---|---|---|
| | app | total | | true | false | total | |
| app-324 | 10 | 40 | 1,072k | 1 | 109 | 110 | 39 |
| noisy-sounds | 11 | 52 | 360k | 21 | 191 | 212 | 34 |
| app-ca7 | 23 | 55 | 1,447k | 124 | 269 | 393 | 34 |
| app-kQm | 31 | 68 | 4,343k | 88 | 729 | 817 | 32 |
| tilt-mazes | 35 | 77 | 1,106k | 90 | 262 | 352 | 35 |
| andors-trail | 44 | 81 | 80k | 5 | 151 | 156 | 23 |
| ginger-master | 39 | 82 | 3,675k | 81 | 356 | 437 | 42 |
| app-018 | 50 | 98 | 6,512k | 7 | 413 | 420 | 69 |

String constants involved as informal information to predict whether an ICC link is true. We provide an example to explain what information we use. Figure 7 presents a component of an Android app, which creates two intents `i1` and `i2` containing sensitive information (device ID) with different `action` fields for ICC matching. It starts other components using the `startActivity` method which will send the information out. The `action` field of the target component's intent filter is "`android.appwidget.action.APPWIDGET_UPDATE`", which matches that of `i2`, but not `i1`. So `i1` can indeed not establish an ICC link. However, the ICC extractor, which is essentially a String analyzer, fails to analyze the value of variable `actionStr` precisely as a container is involved, and returns a wild card "`(.*)`" as an over-approximation. As a result, we get "`(.*)`" and "`(.*).APPWIDGET_UPDATE`" as the `action` fields of `i1` and `i2`, both of which seem to match that of the intent filter. Thus the static analyzer will report two privacy leaks. However, given the String constant "`APPWIDGET_UPDATE`", it is easy to see the ICC link created by `i2` is more likely to be true as the analyzed value of `actionStr` contains no useful information. Following this intuition, by taking the analyzed values of the `action` fields as inputs, which are concatenations between wild cards and String constants, a neural network can predict the confidences that `i1` and `i2` establish an ICC link are 0.82 and 0.91, respectively. Therefore, just like the example in Section 2, the alarm caused by `i2` will rank first after the feedback.

The neural network for predicting how likely an ICC link holds comes from Zhao et al. [42]. Actually, its structure is the same as the one we described in Section 6. Specifically, it uses a convolutional neural network to get the embedding of related fields in *Intent* and *Intent Filter*, then uses a 2-layer fully-connected neural network to output the probability that an ICC link is true.

*7.1.4 Training the Neural Networks.* As mentioned in Section 6, we use the output of the neural network as the conditional probability when encoding soft evidences as noisy sensors. For the pointer analysis, we use two small benchmarks *cache4j* and *raytracer* as the training programs. There are 1,226 labeled alias tuples in them, which are far fewer than those in the programs we use to evaluate our approach. Similarly, we use the 2-object-sensitivity analysis to label alias tuples.

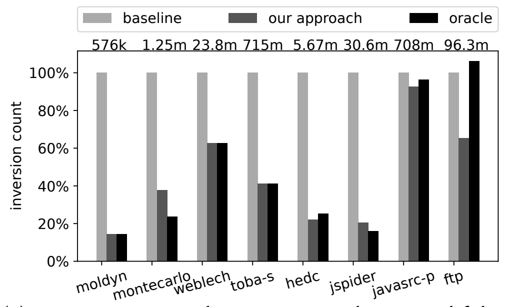
For the taint analysis, we get ICC links from BenchSmall [39], which contains 31 small Android applications downloaded from app markets. To get the ground truth labels of ICC links, we follow the method in a previous work [42]. First, we use IC3 [26] to get ICC links in the benchmarks. That will result in three kinds of links: must links, may links and must not links. Next, we construct may links from must links and must not links by adding imprecision in *Intent* fields. Specifically, we add imprecise patterns "`(.*)`" into `action` and `category` Strings following from the empirical distribution [25]. Finally, we obtain 2,000 ICC links which we know the ground truth labels as the training set. Similarly, we insert ICC links obtained in this way to our evaluated programs.

*7.1.5 Rule Probabilities.* To learn reasonable firing probabilities for all rules, we apply the expectation maximization algorithm [23]. Concretely, we take a small benchmark *cache4j* for the pointer analysis and *VideoActivity* for the taint analysis respectively as the training programs and use the more precise analyses mentioned in section 7.1.1 to label ground truths. As a result, we learn probabilities for all analysis rules including the new rule that derives aliases in the pointer analysis.

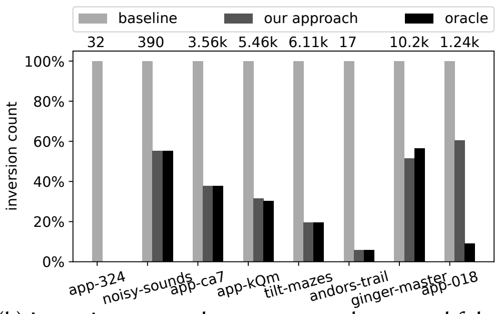
*7.1.6 Configuration of Bayesian Network Inference.* We use the loopy belief propagation algorithm in LibDAI to compute the marginal probabilities on the final Bayesian network that is produced from the Bayesian program analysis with soft evidences. Loopy belief propagation iteratively updates the marginal probabilities. The algorithm terminates either most of the values converge or it reaches a time limit. In the latter case, it may produce less precise results. We set the tolerance in convergence to $10^{-4}$ and the time limit to 57,600 seconds.

## 7.2 Evaluation Results

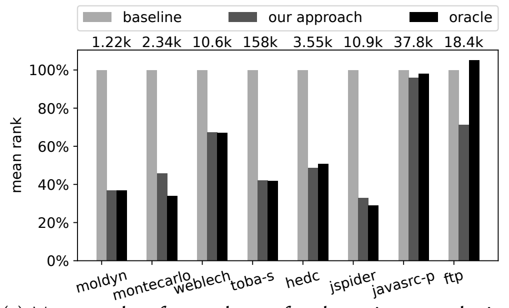
We intend to answer the following four questions with our experiments:



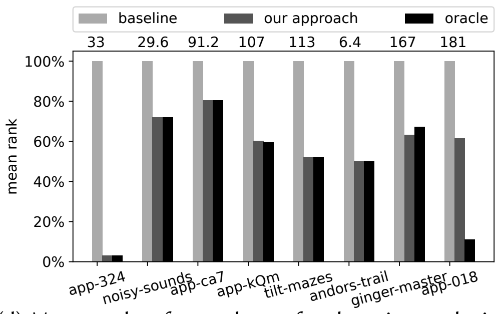
(a) Inversion counts between true alarms and false alarms for the pointer analysis.



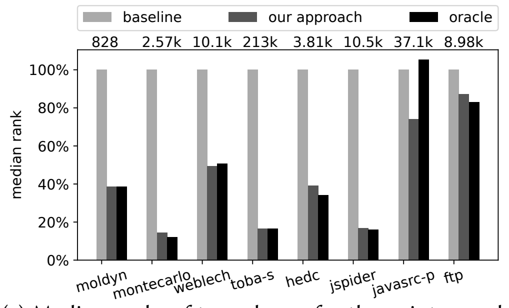
(b) Inversion counts between true alarms and false alarms for the taint analysis.



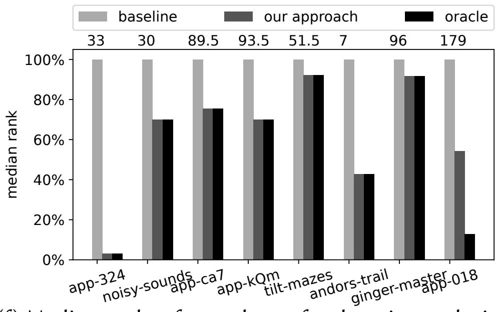
(c) Mean ranks of true alarms for the pointer analysis.



(d) Mean ranks of true alarms for the taint analysis.



(e) Median ranks of true alarms for the pointer analysis.



(f) Median ranks of true alarms for the taint analysis.

Fig. 8. Effectiveness results with three metrics. The three bars in each group from left to right represent the results of the baseline approach, our approach, and the case where oracle labels on aliases are provided as hard evidences, respectively. At the top of each group of bars, we show the metric numbers using the baseline approach. (k = thousand, m = million)

.

(1) *Effectiveness:* How much does NESA improve the alarm rankings of the Bayesian analysis? How far is the effect from the case where all the evidences are observed with oracle labels?
(2) *Efficiency:* How much time does the posterior inference with soft evidences spend? How does it compare to the time that the prior inference without soft evidences spends?
(3) *Comparison to other Bayesian analysis approaches:* How does our approach compare to Bingo which considers user feedback to boost analysis results?
(4) *Sensitivity to training data:* Is the effectiveness of NESA sensitive to the training selection?

*7.2.1 Effectiveness.* Figure 8 shows the effectiveness of our approach in improving rankings of true alarms produced by the pointer analysis and the taint analysis. We use three metrics: inversion count between true alarms and false alarms, mean rank of true alarms, and median rank of true alarms. We compare our approach with two approaches. One is the vanilla Bayesian program analysis without soft evidences, which we referred to as the baseline approach. The other is a Bayesian program analysis where true labels on the soft evidence tuples are provided as hard evidences. The labels come from the more precise analyses we mentioned in section 7.1.1, and we refer to this approach as the oracle approach. We compare against the oracle approach in order to evaluate how well our soft evidences from the neural network work. For each benchmark, at the top, we show the corresponding metric for the baseline approach. The bars show the percentage computed by dividing the metric yielded by an approach by the metric yielded by the baseline approach. The first bar represents the result of the baseline approach, which is always 100%; the second bar shows the result of our approach; the third bar shows the result where the oracle labels on aliases are directly provided as hard evidences.

As we can see, despite training on a small set of data, our approach has significantly improved the ranks of true alarms. In particular, compared to the baseline, it has reduced the inversion count by 55.4%, the mean rank of true alarms by 44.9%, the median rank of true alarms by 58% on average on the pointer analysis. On certain benchmarks, the improvements on these three metrics are as high as 85.6% (on *modlyn*), 67.2% (on *jspider*), and 85.7% (on *montecarlo*). Overall, the improvement in median ranks is more significant than that in mean ranks. This is because there are some true alarms that are not affected by the soft evidences, and their ranks remain low. In terms of absolute numbers, our approach has reduced the mean and median rank of true alarms from 1,220.2 and 827.5 to 450.5 and 320.5 respectively on *moldyn*. Note there are 640 true alarms on *moldyn*.

On the taint analysis, the average improvement on the three metrics is 67.2%, 44.7%, and 37.6% respectively. On *app-324*, we improve the only true alarm's rank from 33 to 1, so the inversion count becomes 0. We take a deeper look into it and find that the soft evidences are correlated to the high-ranking false positives whose posterior probabilities decrease significantly after considering soft evidences. Note that our approach works better on mean ranks this time. It is because on some benchmarks (e.g., *tilt-mazes*, *ginger-master*), the low mean rank is caused by some low-ranked true alarms. These results show that *by encoding informal information as soft evidences, our approach can significantly improve the quality of alarm ranking produced by a Bayesian program analysis.*

Figure 8 also shows that on most benchmarks, the performance of our approach in terms of true alarm ranking is very close to that of using oracle evidence labels as hard evidences. Surprisingly, the inversion count and the mean rank are even significantly better than the oracle approach on some benchmarks (e.g., *ftp*). But the oracle approach performs better in terms of the median rank. This indicates that our approach is more effective in improving the ranks of certain true alarms or reducing the ranks of certain false alarms. Why is this the case? This is due to the approximate nature of static analysis rules. It can lead to false correlations between analysis facts. Sometimes, when a right observation is provided on an analysis fact, the information can generalize to facts that are actually irrelevant according to the concrete semantics. This issue can be mitigated by using better analysis abstractions and rule probabilities. We plan to investigate this in future works.

Figure 9 and 10 show the distribution of true alarms in the rankings by box plot for all benchmarks. From the figures, we can see that for most benchmarks, our approach can cause a great improvement on most of the true alarms' rankings. However, we notice that there are many discrete points in some benchmarks, such as *montecarlo*, *moldyn* and so on. These points are the alarms whose ranks are too far away from the average values. This indicates that for a few true alarms ranked low, our approach fails to rank them higher. This is because a small fraction of soft evidences related to these alarms are predicted in a wrong way.

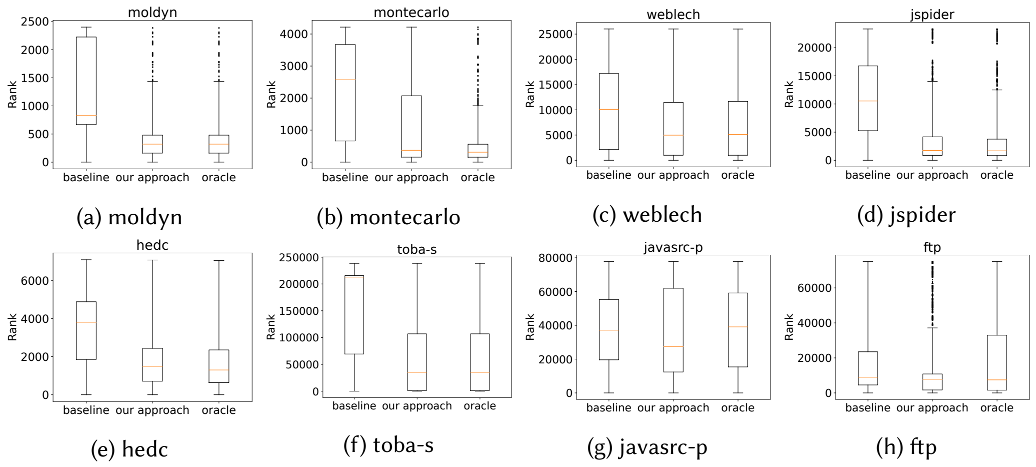


Fig. 9. Comparing ranks of true alarms among baseline, our approach and the case where oracle labels on aliases are provided as hard evidences on the pointer analysis. The $y$-axis shows the ranks of true alarms. The boxes indicate where 25%, 50%, and 75% of true alarms are ranked. The discrete points are alarms whose ranks are too far from the average values.

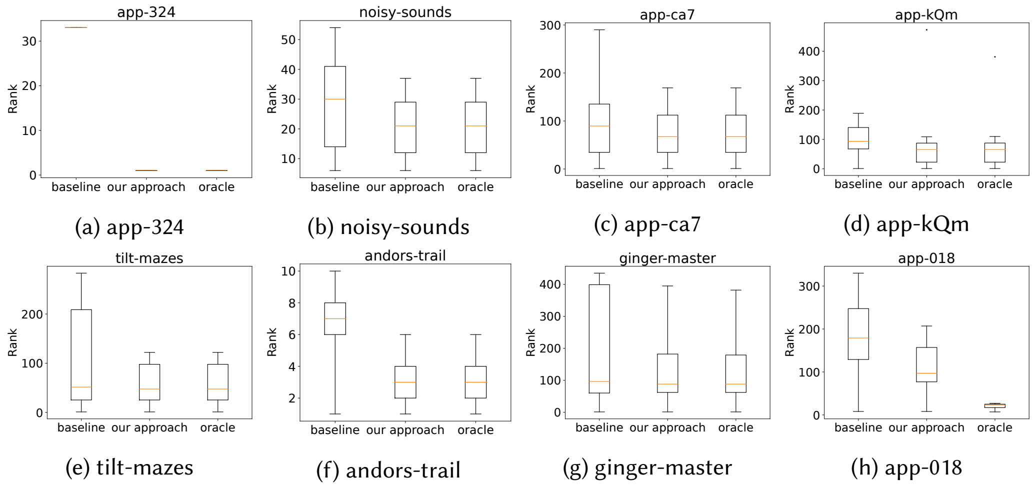


Fig. 10. Comparing ranks of true alarms among baseline, our approach and the case where oracle labels on aliases are provided as hard evidences on the taint analysis. The $y$-axis shows the ranks of true alarms. The boxes indicate where 25%, 50%, and 75% of true alarms are ranked. The discrete points are alarms whose ranks are too far from the average values.

*7.2.2 Inference time.* Table 2a and 2b show the inference time (in seconds) of our approach and the baseline approach without evidences. Overall, our approach takes longer time than the baseline approach on most benchmarks on the pointer analysis. We also observe that on the two largest benchmarks, the times are comparable. And on some benchmarks, like *toba-s* and *ftp*, our approach runs faster than baseline. For the taint analysis, the times of our approach are all comparable with those of baseline. The gap between inference times is understandable. Without evidences,

Table 2. Comparison of inference time between different approaches.

(a) Inference time of the baseline and our approach (in seconds) on the pointer analysis.

| | Baseline | Our Approach |
|---|---|---|
| moldyn | 6.2 | 19.3 |
| montecarlo | 20.1 | 23.4 |
| weblech | 436.2 | 6069.1 |
| jspider | 406.9 | 5868.1 |
| hedc | 292.2 | 1702.7 |
| toba-s | 5721.6 | 3237.7 |
| javasrc-p | 20698.9 | 58000 |
| ftp | 8978.5 | 5321.5 |

(b) Inference time of the baseline and our approach (in seconds) on the taint analysis.

| | Baseline | Our Approach |
|---|---|---|
| app-324 | 1.67 | 1.67 |
| noisy-sounds | 1.17 | 1.34 |
| app-ca7 | 2.15 | 1.6 |
| app-kQm | 5.27 | 6.98 |
| tilt-mazes | 1.46 | 1.32 |
| andors-trail | 0.87 | 0.78 |
| ginger-master | 6.88 | 9.9 |
| app-018 | 8.55 | 9.75 |

(c) Inference time of our approach and the first 200 iterations of Bingo (in seconds).

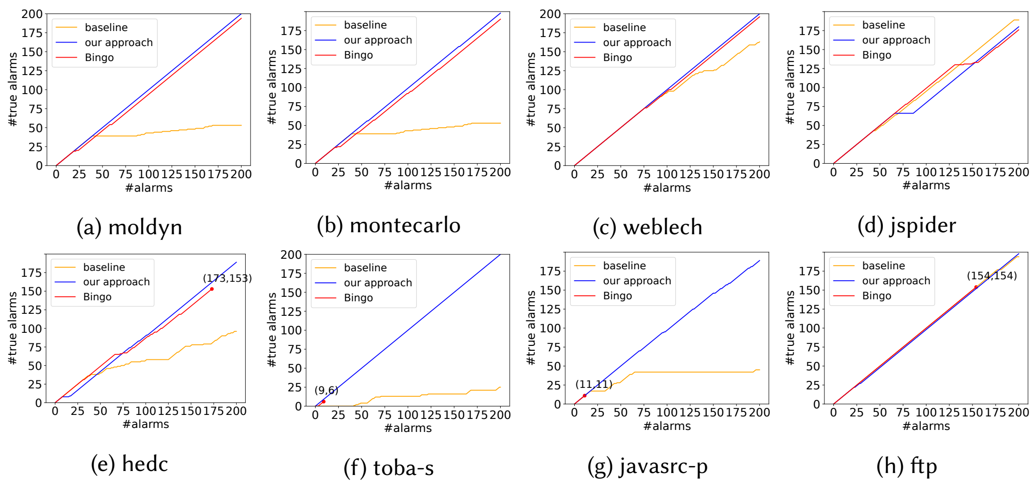
| | Bingo | Our Approach |
|---|---|---|
| moldyn | 2994.2 | 19.3 |
| montecarlo | 23951 | 23.4 |
| weblech | 145858 | 6069.1 |
| jspider | 183615 | 5868.1 |
| hedc | timeout | 1702.7 |
| toba-s | timeout | 3237.7 |
| javasrc-p | timeout | 58000 |
| ftp | timeout | 5321.5 |

the marginal probabilities on a Bayesian network can be simply computed by multiplying the conditional probabilities, which is linear in the size of the network. On the other hand, the general marginal inference problem with evidences is a counting problem. Why is the gap smaller on larger benchmarks? Because the inference time correlates to the fraction of random variables that are being observed on. It would be interesting to explore how to select a subset of soft evidences to balance the effectiveness and efficiency of our approach. The inference time also relies on the convergence state of the loopy belief propagation algorithm. Finally, although our approach takes more time in general, the improvement in true alarm ranks is significant. We consider such trade-off between time and accuracy worthwhile, which is also seen in other Bayesian analysis approaches.

*7.2.3 Comparison with Bingo.* Our approach is orthogonal and complementary to existing Bayesian analysis based approaches as they use different posterior information and there is no conflict in combining them. To compare, we evaluate our approach against Bingo [30], a representative Bayesian analysis approach that uses binary user feedback on alarms for interactive alarm resolution. Briefly, Bingo produces a ranked list of alarms using a Bayesian program analysis, and poses the top uninspected alarm for the user to inspect; after inspecting the alarm, a binary feedback on whether the alarm is true is added as a hard evidence to the analysis, and the ranked list is updated. Compared to our approach, there are mainly two differences. First, the posterior information is different. Second, Bingo is an interactive system that needs to perform Bayesian inference every time a new user feedback comes. While our approach only needs to perform Bayesian inference once as the soft evidences are produced in batch. From a user's perspective, both approaches produce a ranked list of alarms. Our approach generates this list in one shot, while for Bingo, it generates the list dynamically as the user provides feedback.

We compare our approach with Bingo on effectiveness and efficiency on the pointer analysis. For effectiveness, we compare the quality of the aforementioned ranked lists of the two approaches. However, it is too time-consuming for Bingo to produce such a list completely. Hence, we look at the top 200 alarms in the lists and count how many true alarms are in them to evaluate effectiveness. In other words, we only run Bingo up to 200 interactions. Still, Bingo takes a long time to run on larger benchmarks so we set 1 week as the timeout value. We apply a 2-object-sensitivity analysis to pre-compute the labels on the alarms which are in turn used to simulate the user in Bingo.

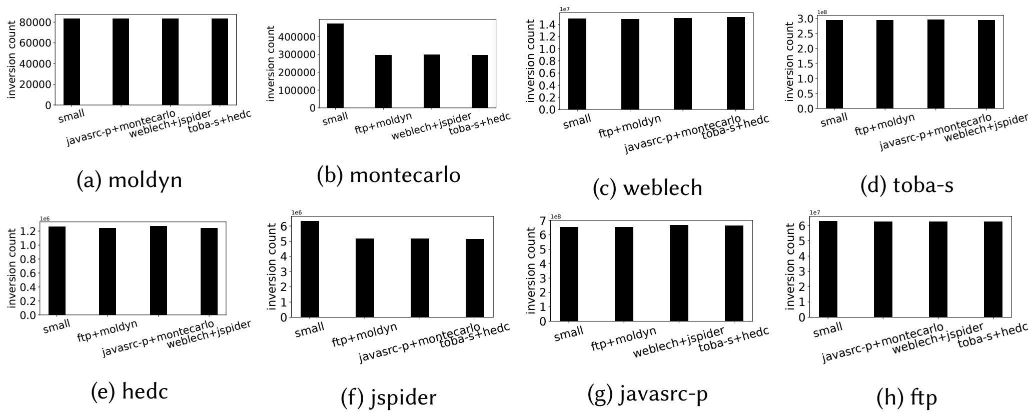
Figure 11 shows the effectiveness results. It shows the number of discovered true alarms as one goes down the ranked list of alarms produced by each approach. We also show the results of the baseline approach where no evidence is considered. As we can see, on smaller benchmarks, the number of true alarms discovered by our approach and Bingo are comparable. Both approaches outperform the baseline significantly. On larger benchmarks, Bingo is only able to run for a few



(a) moldyn (b) montecarlo (c) weblech (d) jspider

(e) hedc (f) toba-s (g) javasrc-p (h) ftp

Fig. 11. The number of true alarms in the top 200 alarms compared with Bingo. Each plot represents the results on a given benchmark on baseline, our approach and Bingo.



(a) moldyn (b) montecarlo (c) weblech (d) toba-s

(e) hedc (f) jspider (g) javasrc-p (h) ftp

Fig. 12. Sensitivity results of inversion counts to training sets. Each plot represents the results on a given benchmark when different training sets are used.

iterations, while our approach is able to produce the full ranked list and outperforms the baseline significantly. For instance, Bingo is only able to produce 9 ranked alarms within one week on *toba-s*.

We further study the inference time of our approach and Bingo on smaller benchmarks. Table 2c shows the inference time of the two in seconds. Note that the time of our approach is that of one turn of inference, which can give a ranking of alarm at one time. On the other hand, the time of Bingo is that of 200 iterations of inference. It is straightforward to see that Bingo takes much more time than our approach, since it needs many iterations of user feedback. On some benchmarks, the time Bingo takes is over 15 times of that our approach takes. Moreover, Bingo cannot terminate within one week on the larger benchmarks, while our approach can terminate in a day.

*In summary, our approach is comparable to Bingo in terms of the effectiveness of improving alarm rankings, but much more efficient.*

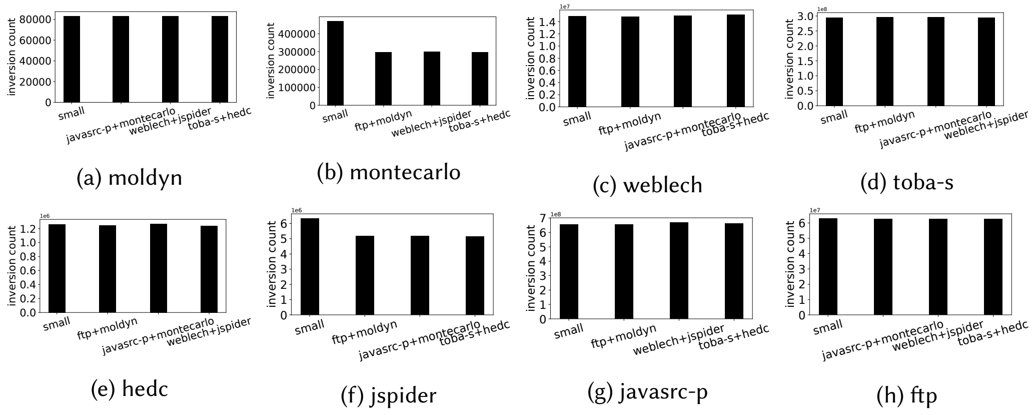


Fig. 13. Sensitivity results of mean ranks to training sets. Each plot represents the results on a given benchmark when different training sets are used.

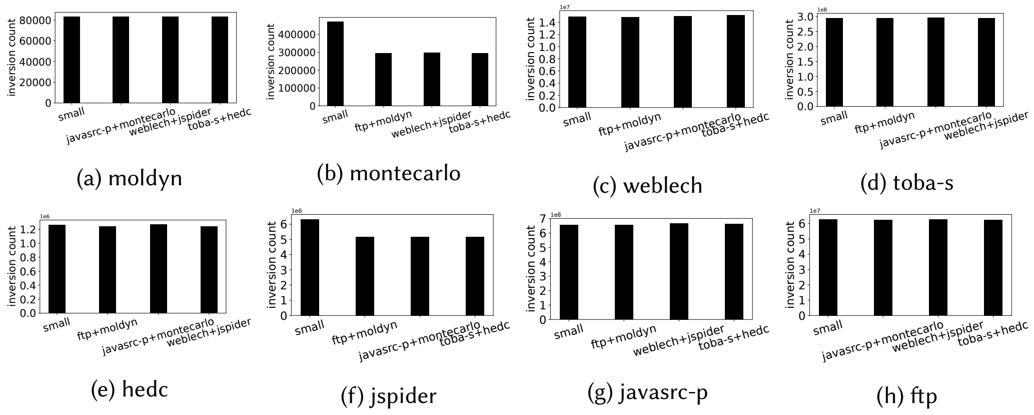


Fig. 14. Sensitivity results of median ranks to training sets. Each plot represents the results on a given benchmark when different training sets are used.

*7.2.4 Sensitivity to Training Data.* To test how sensitive the effectiveness of our approach is to the training data, we run our approaches with different training programs on the pointer analysis. First, we divide the eight Java benchmarks into four groups evenly. Then, we perform four experiments where we use one group for training and the other three for testing. For each program, along with the previous experiment where the two small programs serve as the training set, there are four groups of results using different training sets.

Figure 12, Figure 13, and Figure 14 show the sensitivity results in inversion count, mean ranks, and median ranks respectively. We observe two trends across these results. First, for most benchmarks, the results have improved when larger programs are used for training. Second, the results do not vary drastically when different training sets are used. These two trends indicate that our approach is robust in terms of the choice of training set and improves when more data is given.

## 8 Extension On Incorporating Dynamic Information

Besides informal information, our soft evidence mechanism described in Section 5 can be applied to incorporate other kinds of noisy information to boost analysis accuracy. In this section, we use dynamic information from test runs as an example to demonstrate this ability. DynaBoost [4] has demonstrated the possibility of leveraging dynamic information to improve static analysis results, but it does not provide a systematic way to encode such information in a Bayesian analysis. In fact, it can be seen as an instance of our technique. In this section, we describe first how to incorporate dynamic information from test executions using soft evidences and then evaluate it empirically with an interval analysis and a taint analysis for C programs.

### 8.1 Incorporating Dynamic Information via Soft Evidences

Since typically static analyses over-approximate and test information under-approximates, one can apply test information to confirm analysis facts but not to refute them. However, the more test runs where an analysis fact is not observed, the less likely it holds. Based on this intuition, whether an analysis fact is observed in a set of test runs can serve as a "noisy sensor" on whether it holds.

We now explain how to incorporate such information using soft evidences introduced in Section 5 and the key is how to establish the conditional probability of an analysis fact being observed in a set of test runs given its truth. Given a fixed static analysis, a program that is sampled from a distribution, a true fact that is uniformly sampled from true analysis facts on the program, and a test input that is sampled from a distribution, let $p$ be the probability of the fact being observed in the test run. Given an analysis fact, let $t$ denote whether it is true. Given $N$ test runs with sampled inputs, let $t'$ denote whether the fact is observed, which is the "noisy sensor" of $t$. Then the probability of a true analysis fact not being observed is $(1-p)^N$, while that of a false analysis fact is 1. That is, $P(\neg t' \mid t) = 1 - P(t' \mid t) = (1-p)^N$ and $P(\neg t' \mid \neg t) = 1 - P(t' \mid \neg t) = 1$.

If we have observed $t$ on one test execution, then we make a positive observation on $t'$. By the equation in Section 5.2, the Bayes factor $k = \frac{P(t'|t)}{1-P(\neg t'|\neg t)}$ goes to infinity since the denominator is 0. That is to say, $P(t \mid t') = 1$, i.e. if we witness the program fact by dynamic analysis, then it is confirmed to be true. On the other hand, if we fail to observe $t$ in all executions, then we perform a negative observation on $t'$. At this time, $k = \frac{P(\neg t'|t)}{1-P(t'|\neg t)} = (1-p)^N < 1$. The posterior probability of $t$ will decrease. Moreover, the more test inputs we have, the more we believe $t$ is actually false.

We have not discussed how the hyper-parameter $p$ is set: It can be set manually based on expert knowledge or calibrated on a set of programs where the truths of certain facts are known.

DynaBoost [4] follows a similar spirit but is implemented in a less principled way[4]. Briefly, for each analysis fact $t$, it creates a "noisy sensor" $t'$ and a probabilistic ground rule $t' : -t\ q$ where the probability $q$ varies based on the test information. If the analysis fact is observed, then $t'$ gets a positive hard evidence and $q$ is set to 1. Otherwise, $t'$ gets a negative hard evidence and $q$ is set through a magic function such that $q$ decreases as the number of test cases increases.

### 8.2 Empirical Evaluation

We evaluate our technique of incorporating dynamic information using an interval analysis and a taint analysis on a suite of C programs. We follow the same experiment setting of DynaBoost [4] including benchmark selection. In particular, the test information is used to boost an interactive Bayesian analysis where the user inspects the most likely alarm and the ranking updates upon user feedback. We apply Sparrow [27] as the static analysis engine, and DFSan [36] as the dynamic analysis tool to provide feedback on dataflow facts.

[4]The mechanism was not described in the paper, which we came to know by reading the code and contacting the authors.

Table 3. Experiment result of the dynamic information experiment."Analysis" stands for the analysis type (interval overflow or taint analysis). "#Alarms" and "#Bugs" stand for the number of alarms reported by the static analyzer and the number of true alarms respectively. "Init" stands for the average rank of true alarms without user feedback, while "Iters" stands for the number of iterations required by a user to identify all true alarms. For both metrics, smaller values are better.

| | Analysis | #Alarms | #Bugs | Our Method | | DynaBoost | | Baseline | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Init | Iters | Init | Iters | Init | Iters |
| bc | Interval | 535 | 2 | 121 | 47 | 121 | 48 | 327 | 113 |
| cflow | Interval | 805 | 1 | 172 | 21 | 173 | 21 | 356 | 105 |
| grep | Interval | 912 | 1 | 143 | 63 | 144 | 66 | 714 | 378 |
| gzip | Interval | 344 | 8 | 234.75 | 235 | 230 | 235 | 229.5 | 326 |
| libtasn1 | Interval | 357 | 1 | 5 | 14 | 5 | 14 | 113 | 6 |
| patch | Interval | 502 | 1 | 58 | 14 | 58 | 14 | 227 | 33 |
| readelf | Interval | 882 | 1 | 299 | 87 | 300 | 88 | 111 | 37 |
| sed | Interval | 819 | 1 | 314 | 70 | 316 | 70 | 469 | 196 |
| sort | Interval | 715 | 1 | 460 | 105 | 461 | 106 | 479 | 174 |
| tar | Interval | 1,369 | 1 | 250 | 88 | 253 | 91 | 602 | 220 |
| optipng | Taint | 67 | 1 | 3 | 4 | 3 | 4 | 2 | 4 |
| latex2rtf | Taint | 13 | 2 | 6.5 | 5 | 6.5 | 5 | 5.5 | 4 |
| shntool | Taint | 23 | 6 | 15.3 | 21 | 15.3 | 21 | 15.2 | 21 |
| **Average** | | | | 160.1 | 59.5 | 160.4 | 60.2 | 280.8 | 124.2 |

Table 3 summarizes the result. We compare our technique with two approaches, DynaBoost [4]and a baseline which is the vanilla Bayesian analysis without any dynamic information. We consider two metrics. The "Init" column is the average rank of true alarms without user feedback, while the "Iters" column represents the number of iterations a user needs to inspect all true alarms. Our approach has a 43.0% and 52.1% improvement on the two metrics compared against the baseline. On the other hand, the performance of our approach is comparable to that of DynaBoost, which confirms the observation that DynaBoost can be seen as an instance of our framework.

## 9 Related Work

The most relevant works are literature in Bayesian program analyses and program analyses that integrate a neural component. Compared to the former, ours is the first to apply soft evidences to integrate a neural network. Compared to the latter, our approach offers a general and systematic way to incorporate information from neural network in program analyses. We also survey probabilistic programming languages supporting soft evidences and neurosymbolic programming methods.

*Bayesian Program Analyses.* Mangal et al. [17] first showed the power of Bayesian program analysis by learning from user feedback to improve analysis precision. The use case is to ask users to inspect a few alarms and suppress alarms that are similar to the ones that users label as false. They also work with program analyses that are expressed in Datalog. They identify analysis rules that are likely to be the root causes of false alarms and convert them into soft rules by adding weights. The resulted Bayesian analysis is expressed in a Markov Logic Network [31]. The user feedback is also added as rules to the formulation. Finally, new alarms are generated through maximum a posteriori probability inference on the Markov Logic Network. Raghothaman et al. [30] also proposed an approach to improve analysis precision using user feedback. The main difference is that their approach produces a ranking of alarms based on the marginal probabilities. In addition, they did not explicitly re-formulate the analysis in a probabilistic logic program, but directly cast the analysis derivations into a Bayesian network. Chen et al. [4] further improved the approach by additionally considering dynamic information from test runs. Compared to these approaches, our external information to boost the analysis is different. Moreover, our approach is the first to incorporate noisy external information systematically using soft evidences.

*Static Analyses with Neural Networks.* Koc et al. [11] studied applying different machine learning models as a post-processor to classify true alarms and false alarms produced by a given analysis. In particular, they apply a recurrent neural network and graph neural network that take program slices as inputs. Similarly, Zhao et al. [42] applied a neural network to classify the results of an analysis that discovers communication links between Android applications. The main differences are 1) they use the must results of the analysis as data to train how to classify the may results, and 2) the neural network is specially designed to take the abstract domain of the analysis as its input. Wang et al. [38] proposed to use a neural network to directly find bugs. Compared to conventional analyses, it is better at identifying bugs based on syntactical patterns such as variable misuses, but it is worse at finding bugs that need reasoning about program semantics, such as memory errors. Moreover, it offers no soundness guarantee. Compared to these works, ours is the first to integrate a static analysis with a neural network in a systematic and general manner.

*Soft Evidences in Probabilistic Programming.* We are not aware of any probabilistic logic language supporting soft evidences, but many non-logic probabilistic languages offer low-level mechanisms to support them. Concretely, they offer language constructs to re-weight the probability of a given execution path [3, 8, 19, 37]. It takes effort to reformulate our analysis with soft evidences using these languages. More importantly, their built-in inference engines cannot scale to our problems.

*Neurosymbolic Programming.* Finally, we compare to works on neurosymbolic programming, which aim to incorporate neural networks into symbolic reasoning systems. They typically augment a probabilistic programming language with neural network outputs as program inputs, but they do not support incorporating neural network outputs as soft evidence. DeepProblog [18] extends Problog [29], a probabilistic logic programming language, with neural network outputs as random variables. Scallop [15, 16] extends Datalog with probabilities and neural networks. It can reason with inputs like images and natural language texts. Its main advantage is to enable end-to-end training for the symbolic reasoning part and the neural network part. Neither of the languages can be applied to add neural network outputs as evidences for backward probability inference.

## 10 Conclusion and Future Work

We have proposed NESA, a general framework to incorporate informal information in a Datalog-based program analysis systematically. In particular, our approach uses a neural network to estimate how likely a program fact holds based on informal information such as variable names. Our approach adds rules to the analysis to derive these facts if needed, converts the analysis into a probabilistic one, and then encodes neural network outputs as virtual evidences which are essentially "noisy sensors". This quantifies the uncertainty in the information without breaking the soundness. The probabilistic analysis is then transformed into a Bayesian network, and alarms are ranked based on their marginal probabilities. We have demonstrated the effectiveness of our approach by augmenting a pointer analysis with variable name information, and a taint analysis with ICC information.

For future work, richer sources of informal information can be applied. In general, as long as the informal information can be used to predict whether an analysis fact holds, then it can be incorporated by our approach. For example, an interesting source would be to apply a language model to extract information from the comments. Tan et al. [35] have used comments to extract pre- and post-condition annotations related to interrupts. While it uses comments to predict analysis facts, it still requires manual validation. With our approach, such information can be systematically incorporated as soft evidence, potentially eliminating the need for manual confirmation.

## Data-Availability Statement

The artifact of our paper is available at Zenodo [14]. It includes all the source code, scripts, data and statistics in our experiments. All results in our experiments can be reproduced. It also includes a reusability guide to encourage further exploration.

## Acknowledgments

We would like to thank the anonymous reviewers for their feedback and comments. We would also like to thank the authors of Dynaboost [4] for their detailed introduction to their system. The work is supported by the National Natural Science Foundation of China under Grant No. 62172017.

## References

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29.

[3] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6. http://jmlr.org/papers/v20/18-403.html

[4] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1154–1165. doi:10.1145/3468264.3468626

[5] Adnan Darwiche. 2009. *Probability Calculus.* Cambridge University Press, 27–52. doi:10.1017/CBO9780511811357.004

[6] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: semantics-based detection of Android malware through static analysis *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 576–587. doi:10.1145/2635868.2635869

[7] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory Pract. Log. Program.* 15, 3 (2015), 358–401. doi:10.1017/S1471068414000076

[8] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2023-4-14.

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[10] Tomi Janhunen. 2004. Representing Normal Programs with Clauses. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, Ramón López de Mántaras and Lorenza Saitta (Eds.). IOS Press, 358–362.

[11] Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. 2019. An Empirical Assessment of Machine Learning Approaches for Triaging Reports of a Java Static Analysis Tool. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 288–299. doi:10.1109/ICST.2019.00036

[12] Triet Huynh Minh Le, Hao Chen, and Muhammad Ali Babar. 2021. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3 (2021), 62:1–62:38.

[13] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: detecting inter-component privacy leaks in Android apps *(ICSE '15)*. IEEE Press, 280–291.

[14] Tianchi Li and Xin Zhang. 2025. *Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences (Paper Artifact)*. doi:10.5281/zenodo.14942368

[15] Ziyang Li, Jiani Huang, Jason Liu, Felix Zhu, Eric Zhao, William Dodds, Neelay Velingker, Rajeev Alur, and Mayur Naik. 2024. Relational Programming with Foundational Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 9 (Mar. 2024), 10635–10644. doi:10.1609/aaai.v38i9.28934

[16] Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A Language for Neurosymbolic Programming. *Proc. ACM Program. Lang.* 7, PLDI, Article 166 (jun 2023), 25 pages. doi:10.1145/3591280

[17] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 462–473. doi:10.1145/2786805.2786851

[18] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. DeepProbLog: Neural Probabilistic Logic Programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 3753–3763. https://proceedings.neurips.cc/paper/2018/hash/dc5d637ed5e62c36ecb73b654b05ba2a-Abstract.html

[19] Vikash Mansinghka, Daniel Selsam, and Yura N. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099 (2014). arXiv:1404.0099 http://arxiv.org/abs/1404.0099

[20] Theofrastos Mantadelis and Gerda Janssens. 2010. Dedicated Tabling for a Probabilistic Setting. In *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK (LIPIcs, Vol. 7)*, Manuel V. Hermenegildo and Torsten Schaub (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 124–133. doi:10.4230/LIPIcs.ICLP.2010.124

[21] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 1–11.

[22] Joris M. 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. http://www.jmlr.org/papers/volume11/mooij10a/mooij10a.pdf

[23] Todd K Moon. 1996. The expectation-maximization algorithm. *IEEE Signal processing magazine* 13, 6 (1996), 47–60.

[24] Mayur Naik. 2011. Chord: A versatile platform for program analysis. In *Tutorial at ACM Conference on Programming Language Design and Implementation*.

[25] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. *SIGPLAN Not.* 51, 1 (jan 2016), 469–484. doi:10.1145/2914770.2837661

[26] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 77–88. doi:10.1109/ICSE.2015.30

[27] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 229–238. doi:10.1145/2254064.2254092

[28] Judea Pearl. 1989. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann.

[29] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, Manuela M. Veloso (Ed.). 2462–2467. http://ijcai.org/Proceedings/07/Papers/396.pdf

[30] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 722–735. doi:10.1145/3192366.3192417

[31] Matthew Richardson and Pedro M. Domingos. 2006. Markov logic networks. *Mach. Learn.* 62, 1-2 (2006), 107–136. doi:10.1007/s10994-006-5833-1

[32] Fabrizio Riguzzi and Theresa Swift. 2018. A survey of probabilistic logic programming. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM / Morgan & Claypool, 185–228. doi:10.1145/3191315.3191319

[33] Taisuke Sato. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, Leon Sterling (Ed.). MIT Press, 715–729.

[34] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: value-flow-based precise code embedding. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 233:1–233:27.

[35] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/1985793.1985796

[36] The Clang Team. 2020. DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html

[37] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank D. Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*, Tom Schrijvers (Ed.). ACM, 6:1–6:12. doi:10.1145/3064899.3064910

[38] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 137:1–137:27. doi:10.1145/3428205

[39] Jiwei Yan, Shixin Zhang, Yepang Liu, Xi Deng, Jun Yan, and Jian Zhang. 2023. A Comprehensive Evaluation of Android ICC Resolution Techniques. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. doi:10.1145/3551349.3560420

[40] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 239–248. doi:10.1145/2594291.2594327

[41] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 954–982.

[42] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018. Neural-augmented static analysis of Android communication. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 342–353. doi:10.1145/3236024.3236066

Received 2024-10-16; accepted 2025-02-18