Building effective program analysis tools is a challenging endeavor: analysis designers must balance multiple competing objectives, including scalability, fraction of false alarms, and the possibility of missed bugs. Not all of these design decisions are optimal when the analysis is applied to a new program with different coding idioms, environment assumptions, and quality requirements. Furthermore, the alarms produced are typically accompanied by limited information such as their location and abstract counter-examples. We present a framework DFFLOG that fundamentally extends the deductive reasoning rules that underlie program analyses with numerical weights. Each alarm is now naturally accompanied by a score, indicating quantities such as the confidence that the alarm is a real bug, the anticipated severity, or expected relevance of the alarm to the programmer. To the analysis user, these techniques offer a lens by which to focus their attention on the most important alarms and a uniform method for the tool to interactively generalize from human feedback. To the analysis designer, these techniques offer novel ways to automatically synthesize analysis rules in a data-driven style. DFFLOG shows large reductions in false alarm rates and missed bugs in large, complex programs, and it advances the state-of-the-art in synthesizing non-trivial analyses.

1 Introduction

The ideal program analysis tool successfully flags the most serious bugs, produces no false alarms, and gracefully scales to arbitrarily large codebases. The challenges involved in building these tools always force a compromise between these competing requirements. Analysis designers strive to identify highly scalable approximations that still produce results with acceptable accuracy, while analysis users demand greater control over the number of alarms produced, ways to prioritize reports that are most likely to be relevant and represent real bugs, and would like the tool to learn these preferences from past interaction.

Traditional approaches to program reasoning rely on deductive techniques where these limitations manifest either as unsound or as incomplete analysis rules. Applied to a large codebase, these analysis rules produce false alarms in statistically regular ways. Furthermore, because multiple alarms share portions of their derivation trees, it follows that they are correlated in their ground truth, an observation which has also been exploited by previous research in improving analysis accuracy [24, 27, 37]. In this paper, we draw inspiration from the large body of research reconciling logical reasoning with machine learning, which has appeared variously as work on probabilistic databases [8, 11], inductive logic programming [28], statistical relational learning [12], and probabilistic programming (e.g. [10] and [13]).
We present DIFFLOG, a framework by which we associate weights with individual deduction rules of the analysis. The notion of *semiring provenance* from databases [14, 7] provides a mechanism to use these weights to associate numerical values with the conclusions of the analysis. Depending on the choice of the underlying semiring, these quantities could have different interpretations, such as the confidence that the alarm represents a real bug, or the expected severity or relevance of the alarm to the user.

*For the analysis user,* the confidence values we output provide a metric by which to rank the alarms and focus on those which are most likely to be true. Furthermore, the system can naturally update the confidence values by conditioning on user feedback. The information gained from repeated rounds of human interaction allows the resulting program analysis system to discover many more bugs and have greatly improved accuracy compared to non-interactive systems.

*For the analysis designer,* DIFFLOG provides a way to exploit numerical techniques such as Newton’s method and gradient descent and automatically synthesize analyses from training corpora. This also raises the intriguing possibility of automatically tuning the precision, recall, and scalability of an analysis sketch based on its performance on a repository of training data. In preliminary experiments, we show that numerical learning techniques synthesize several non-trivial program analyses significantly faster than previous state-of-the-art methods.

2 Overview of the Framework

We describe our technique using the example of Andersen’s analysis [2] shown in Figure 2a. We express the analysis using Datalog, a popular formalism to declaratively specify complex program reasoning algorithms [34, 36, 5, 4, 26, 3]. A Datalog program is a collection of rules, each of the form \( h \leftarrow b_1, b_2, \ldots, b_k \), where each component, \( h, b_1, b_2, \ldots, b_k \), is called a *literal*. The free variables, \( p, q, \ldots \), are implicitly universally quantified, and \( \leftarrow \) is interpreted as implication. For example, the rule \( R_3 \) may be read as “If the program contains the statement \( p \leftarrow \star q \), and \( q \) may point to \( r \), and \( r \) may point to \( s \), then \( p \) may point to \( s \).” The meaning of a Datalog program is defined as a fixpoint: to apply the analysis, one iteratively accumulates conclusions until no further tuples can be derived.

Now consider the program of Figure 2b. We show a portion of the reasoning induced by applying Andersen’s analysis in Figure 3. Note that the variable \( a \) initially points to the memory location \( b_1 \), and
We start by analyzing with the probability \( p \) of the analysis with the probability of being true. For example, if we associate each rule of Andersen’s idea is to associate each rule \( R \) feedback, and automatically infer that \( pt \) the results of the analysis, and indicates that \( pt \) tuple \( pt \) misfiring situation a

However, because the analysis is flow-insensitive, it disregards the order of program statements, and later to the memory location \( b2 \). The assignment \( k := *a \) can therefore only result in \( k \) pointing to \( c3 \). However, because the analysis is flow-insensitive, it disregards the order of program statements, and concludes that \( k \) may also point to \( c1 \) and \( c2 \).

The erroneous conclusions arise because the rule \( R3 \) is incomplete: even if all of the hypotheses, \( p := *q, pt(q,r), \) and \( pt(r,s) \), are true, it is possible for the conclusion \( pt(p,s) \) to be spurious. We call this situation a misfiring of the rule \( R3 \). Observe that since both \( pt(k, c1) \) and \( pt(k, c2) \) share the intermediate tuple \( pt(a, b1) \), they are correlated, in an informal sense, in their ground truth. Suppose the user examines the results of the analysis, and indicates that \( pt(k, c1) \) is false. Is it possible for us to generalize from this feedback, and automatically infer that \( pt(k, c2) \) is also likely to be false?

Note that it is difficult to characterize this correlation in a purely deductive manner: any of the rule instances leading to the faulty conclusion could have misfired, and \( \neg pt(k, c1) \) does not logically entail \( \neg pt(k, c2) \). However, on large programs, rules misfire in statistically regular ways. Therefore, our first idea is to associate each rule \( R \) with a probability \( p_R \) of correctly firing; we postulate that each instance of \( R \) fires independently and with identical probability \( p_R \). We can now associate individual conclusions of the analysis with the probability of being true. For example, if we associate each rule of Andersen’s analysis with the probability \( p_R = 0.9 \), and treat each input tuple as being known with certainty, then

\[
\Pr(pt(k, c2)) = Pr(R3(k, a, b1, c2) \cap R1(b1, c2) \cap R1(a, b1)) = 0.9^3 = 0.729.
\]

The idea is that alarms with higher probability are more likely to be true than alarms with low probability,
and therefore guide the user towards the real bugs in their program. Observe that we have also automatically obtained a way to generalize from user feedback, i.e. by computing conditional probabilities:

$$\Pr(pt(k, c2) | \neg pt(k, c1)) = \frac{\Pr(pt(k, c2) \cap \neg pt(k, c1))}{\Pr(\neg pt(k, c1))} = 0.51 \ll \Pr(pt(k, c2)).$$

(2)

Observe that the unrelated alarm $pt(k, c3)$ is unaffected: $\Pr(pt(k, c3) | \neg pt(k, c1) = \Pr(pt(k, c3)) = 0.729$.

The key challenge behind formalizing this intuition is to succinctly describe how each output tuple came to be derived. We do this by adapting the concept of semiring provenance from databases [14, 7]. Briefly, we first fix a semiring $(K, +, \times, 0_K, 1_K)$, and each instance $g$ of a rule with a value $k_g \in K$. For our example of computing probabilities, $k_g$ is the event that $g$ has correctly fired, the semiring operations $+$ and $\times$ represent union and intersection respectively—$k + k' = k \cup k'$, and $k \times k' = k \cap k'$—and $0_K = \emptyset$ and $1_K = \Omega$, the set of outcomes of the probability space. The provenance $k_t$ of a tuple $t$ is given by:

$$k_t = \sum_{\tau} \prod_{g \in \tau} k_g,$$

(3)

where $\tau$ ranges over all derivation trees whose conclusion is $t$, and $g \in \tau$ ranges over all rule instances occurring in $\tau$. The probability of a tuple $t$ is now given by the probability of its associated event: $\Pr(t) = \Pr(k_t)$. We have implemented this idea in recent work [32] to obtain an interactive program analysis system. The user repeatedly inspects the alarm with highest confidence, and the system incorporates their feedback and reranks the remaining alarms. We present an excerpt of our experimental results in Table 1. On average, across two analyses—a datarace analysis applied to Java programs, and a taint analysis applied to Android apps—and on a suite of 16 real-world benchmark programs, the user has to inspect 44.2% fewer alarms than the baseline to discover all bugs.

Unfortunately, each conclusion of a Datalog program may have many (possibly even infinitely many) derivation trees. Therefore, the main technical difficulty in instantiating our framework is the algorithmic complexity of Equation 3. The system we describe in Table 1 tackles this problem by only performing approximate marginal inference in a Bayesian network along with aggressive constraint pruning. Another approach is to change the underlying semiring, for example to the max-times semiring. Here, $K$ is the closed interval of truth values $[0, 1]$, and the semiring operations are defined as $k + k' = \max(k, k')$ and $k \times k' = kk'$, $0_K = 0$ and $1_K = 1$. Equation 3 can be readily evaluated using efficient algorithms such as seminaive evaluation. The most important motivator for this semiring is that the gradient of the output value with respect to the rule weights can be efficiently computed, so that the task of synthesizing rule weights, and even the analysis itself, can now be performed by popular numerical techniques such as gradient descent and Newton’s method. In preliminary experiments that we present in Table 2, we show that this technique greatly outperforms the state-of-the-art in learning Datalog programs such as Andersen’s analysis and mod/ref analysis.

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Rules</th>
<th>Program</th>
<th>Size LOC</th>
<th>Tuples</th>
<th>Clauses</th>
<th>Alarms Total</th>
<th>Bugs</th>
<th>DIFFLOG</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datalog</td>
<td>102</td>
<td>FTP Server</td>
<td>152K</td>
<td>2067K</td>
<td>2182K</td>
<td>522</td>
<td>75</td>
<td>103</td>
<td>368</td>
</tr>
<tr>
<td>Taint</td>
<td>62</td>
<td>AndorsTrail</td>
<td>81K</td>
<td>13K</td>
<td>72K</td>
<td>156</td>
<td>7</td>
<td>14</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 1: Experimental performance of an interactive program analysis system based on DIFFLOG [32]. The column **Rules** measures the number of analysis rules. The columns labelled **Last True Rank** indicates the number of alarms that the user needs to inspect before discovering all bugs in the program.
M. Raghothaman, S. Kulkarni, R. Zhang, X. Si, K. Heo, W. Lee, and M. Naik

<table>
<thead>
<tr>
<th>Program</th>
<th>Relations</th>
<th>Rules</th>
<th>Time (in s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input</td>
<td>Output</td>
<td>Synthesized</td>
</tr>
<tr>
<td>Andersen</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Mod/ref</td>
<td>7</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2: Performance of DIFFLOG-based gradient descent in synthesizing program analyses, compared to two state-of-the-art combinatorial algorithms, ALPS [?], and Zaatar [1].

3 Challenges and Opportunities

Interpretability and scalability. The most important challenge in instantiating DIFFLOG is choosing a semiring so that: (a) the output values come with useful guarantees and are readily interpretable by programmers, and (b) Equation 3 can be evaluated efficiently on real-world codebases. One option we discussed was using the min-times semiring so that efficient Datalog evaluation algorithms can be adapted in a straightforward way. Connecting this to other well-studied problems such as min-cuts is an exciting direction of future research.

Learning. In Section 2, we discussed how the min-times semiring enables fast gradient-descent based learning. In the case of the probabilistic semiring, we may also use the expectation-maximization algorithm to learn rule firing probabilities. The most important problem involving learning is that of invented / hidden predicates. Furthermore, while we have assumed a corpus of labelled training data, human feedback is often inaccurate. Can our learning algorithms operate in unsupervised or weakly supervised settings? On the opposite side, can we build standard candles and large repositories of well-labelled programs?

Rich tool output. We have discussed how we may associate alarms with the probability that they represent true bugs. Programmers are also interested in the severity and relevance of individual alarms. We are investigating the applicability of DIFFLOG to produce this information. Furthermore, instead of a passive model where the user chooses which alarm to inspect, there is great value in actively soliciting feedback from the user on alarms and intermediate conclusions of high value [9, 37]. We are investigating the problem of determining tuples whose ground truth would be most effective in reducing uncertainty in remaining alarms.

4 Related Work

There is a large body of research on using either statistical or interactive methods to overcome the incompleteness of automatic program reasoning tools. In this section, we summarize the most significant of these efforts.

Statistical Error Diagnosis Many previous works have leveraged various kinds of program features to prioritize likely static analysis alarms. The z-ranking algorithm [22, 21] applies a statistical technique called z-test to rank alarms based on the observation that static analysis alarms are highly correlated by code locality. In addition, syntactic and analysis-derived features [18, 35, 20] have been employed to prioritize alarms more likely to be true. Our work can be seen as explaining why these techniques tend to work. Correlations of alarms that share portions of their derivation graph are naturally captured by our Bayesian network.
Data-driven Program Analysis  Recently, data-driven approaches have been taken to to find cost-effective abstractions [15, 30, 16, 6, 17], predicting likely types [33, 19], mining likely specifications and finding anomalies [29, 25, 23].

Human-in-the-loop System  These approaches combines user guidance with automated reasoning. Eugene re-classifies alarms at each round of user feedback [27]. URSA [37] searches for the root causes of false alarms in Datalog derivation graphs and asks the user if they can be refuted. To find inductive invariants, Ivy [31] graphically visualizes counterexamples to induction and let the user guide the generalization.

References


