# DIFFLOG: Beyond Deductive Methods in Program Analysis

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

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

Richard Zhang
University of Pennsylvania
rmzhang@cis.upenn.edu

Xujie Si
University of Pennsylvania
xsi@cis.upenn.edu

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

Woosuk Lee
University of Pennsylvania
woosuk@cis.upenn.edu

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

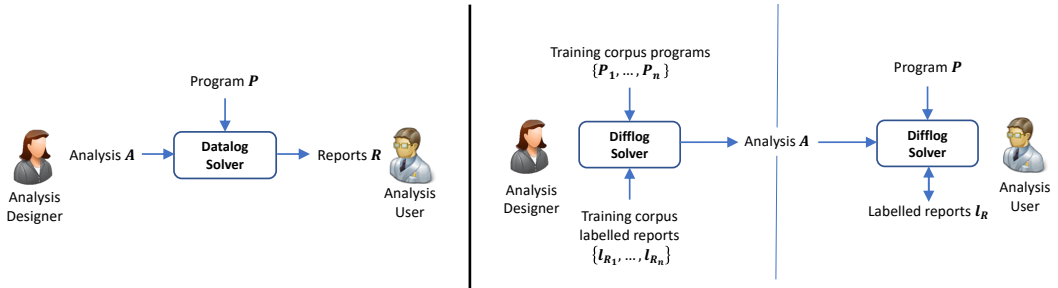
Building effective program analysis tools is a challenging endeavor: analysis designers must balance multiple competing objectives, including scalability, fraction of false alarms, and the possibility of missed bugs. Not all of these design decisions are optimal when the analysis is applied to a new 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 DIFFLOG 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. DIFFLOG 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 [25, 28, 39]. 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 [9, 12], inductive logic programming [29], statistical relational learning [13], and probabilistic programming (e.g. [11] and [14]).

Submitted to:
ML4P 2018




(a) Existing approach. (b) Our approach.

Figure 1: Existing approaches require the analysis designer to completely specify an analysis whose reports have to be inspected in their entirety by the analysis user. In contrast, DIFFLOG uses richly labelled data to satisfy both the needs of the analysis designer and user. For the designer, it can learn the analysis from a partially labelled training corpus. For the analysis user, it generates labelled reports, given the analysis and the program to be analyzed.

We present DIFFLOG (Figure 1), a framework by which we associate weights with individual deduction rules of the analysis. The notion of *semiring provenance* from databases [15, 8] provides a mechanism to use these weights to associate numerical values with analysis conclusions. 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 [35, 38, 6, 4, 27, 3]. A Datalog program is a collection of rules, each of the form $h :\!-\, b_1, b_2, \ldots, b_k$, where each component, $h$, $b_1$, $b_2$, …, $b_k$, is called a *literal*. The free variables, $p$, $q$, …, are implicitly universally quantified, and $:\!-$ is interpreted as implication. For example, the rule $R_3$ may be read as "If the program contains the statement $p := *q$, 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.

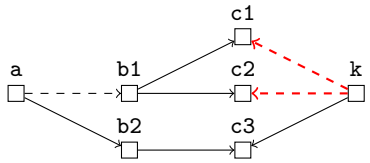
Now consider the program of Figure 2b. We show a portion of the reasoning induced by applying Andersen's analysis in Figure 3. Note that the variable `a` initially points to the memory location `b1`, and

$R_1$ pt$(p,q)$ :– "$p := \&q$"
$R_2$ pt$(p,r)$ :– "$p := q$", pt$(q,r)$
$R_3$ pt$(p,s)$ :– "$p := *q$", pt$(q,r)$, pt$(r,s)$
$R_4$ pt$(r,s)$ :– "$*p := q$", pt$(p,r)$, pt$(q,s)$

(a)

```
a := &b1;
b1 := &c1;
if (*) b1 := &c2;
...
a := &b2;
b2 := &c3;
k := *a;
```

(b)




(c)

Figure 2: Description of Andersen's pointer analysis [2]. The tuple pt$(p,q)$ indicates that the memory location $p$ may point to the memory location $q$. When applied to the program of Figure 2b, it results in the heap graph shown in Figure 2c. The analysis is flow-insensitive, so that it combines the assignment k:=*a with the obsolete conclusion pt(a, b1) to erroneously conclude pt(k, c1) and pt(k, c2).

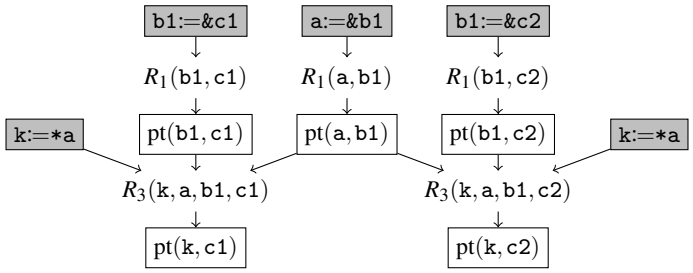



Figure 3: A portion of the derivation graph leading to each conclusion of the analysis. Each rectangle denotes a tuple, where input tuples are shaded in gray, and output tuples are unshaded. Unboxed nodes indicate rule instances, so for example $R_1$(b1, c1) indicates the instance of the rule $R_1$ with $p =$ b1 and $q =$ c1. The arrows indicate logical dependencies between these entities.

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 $R_3$ 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 $R_3$. 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 $p_{R_i} = 0.9$, and treat each input tuple as being known with certainty, then

$$\Pr(\text{pt}(\texttt{k},\texttt{c2})) = \Pr(R_3(\texttt{k},\texttt{a},\texttt{b1},\texttt{c2}) \cap R_1(\texttt{b1},\texttt{c2}) \cap R_1(\texttt{a},\texttt{b1})) = 0.9^3 = 0.729. \quad (1)$$

The idea is that alarms with higher probability are more likely to be true than alarms with low probability,

| Analysis | Rules | Program | Size | | | Alarms | | Last True Rank | |
|---|---|---|---|---|---|---|---|---|---|
| | | | LOC | Tuples | Clauses | Total | Bugs | DIFFLOG | Baseline |
| Datarace | 102 | FTP Server | 152K | 2067K | 2182K | 522 | 75 | 103 | 368 |
| Taint | 62 | AndorsTrail | 81K | 13K | 72K | 156 | 7 | 14 | 48 |

Table 1: Experimental performance of an interactive program analysis system based on DIFFLOG [33]. 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.

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*:

$$\Pr(\mathtt{pt}(\mathtt{k},\mathtt{c2}) \mid \neg\,\mathtt{pt}(\mathtt{k},\mathtt{c1}) = \frac{\Pr(\mathtt{pt}(\mathtt{k},\mathtt{c2}) \cap \neg\,\mathtt{pt}(\mathtt{k},\mathtt{c1})}{\Pr(\neg\,\mathtt{pt}(\mathtt{k},\mathtt{c1}))} = 0.51 \ll \Pr(\mathtt{pt}(\mathtt{k},\mathtt{c2})). \tag{2}$$

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

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 [15, 8]. Briefly, we first fix a semiring $(K,+,\times,0_K,1_K)$, and each instance $g$ of a rule with a token $k_g \in K$. 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:

$$k_t = \sum_{\tau} \prod_{g \in \tau} k_g, \tag{3}$$

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 [33] 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 static 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. As before, each rule $R$ is associated with a *truth value* $p_R \in [0,1]$, and each instance $g$ of $R$ with the same token $k_g = p_R$. 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 now be efficiently evaluated using previously known algorithms such as seminaive evaluation. Furthermore, the partial derivatives, $\partial k_t / \partial p_R$ can also be readily computed: we can now adapt popular numerical optimization techniques such as gradient descent and Newton's method to the task of synthesizing rule weights, and even the analysis itself. 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.

| Program | Relations | | Rules | | Time (in seconds) | | |
|---|---|---|---|---|---|---|---|
| | Input | Output | Synthesized | Candidate | DIFFLOG | ALPS | Zaatar |
| Andersen | 4 | 1 | 4 | 27 | 5.02 | 148 | 295 |
| Mod/ref | 7 | 5 | 10 | 36 | 11.1 | 5307 | Timeout |

Table 2: Performance of DIFFLOG-based gradient descent in synthesizing program analyses, compared to two state-of-the-art combinatorial algorithms, ALPS [36], and Zaatar [1]. Each system was given 3 hours to synthesize the analysis.

# 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 max-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 max-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?

**Rethinking the analysis–user interface.** 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 [10, 39]. 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 which applies statistical methods to syntactic features of the program to determine which alarms are likely to be true [19, 37, 21]. The $z$-ranking algorithm [23, 22] is one prominent example, which uses the $z$-test and ranks alarms according to code locality. By modelling the alarm generation process through Equation 3, our work can be seen as explaining why these techniques tend to work. Statistical methods have also been used to find cost-effective abstractions which are still sufficiently precise to prove the properties of interest [16, 31, 17, 7, 18], predict likely types [34, 20], and mine likely specifications and find anomalies [30, 26, 24]. Human-in-the-loop program analysis systems have largely been based on non-statistical optimization techniques, such as abduction [10], integer linear programming [39], and MaxSMT [28]. They are also often formulated to pinpoint the *root cause* of errors in the analysis [39, 32], which we encode probabilistically in DIFFLOG. Finally, Bielik et al [5] consider the problem of automatically learning a static analyzer from data, but focus on a restricted class of analyses which can be expressed using decision trees over program syntax, as opposed to the deeper properties which we can model using fixpoints.

## Acknowledgments

We thank the anonymous reviewers for insightful comments. This research was supported by DARPA under agreement #FA8750-15-2-0009 and by NSF awards #1253867 and #1526270.

## References

[1] Aws Albarghouthi, Paraschos Koutris, Mayur Naik & Calvin Smith (2017): *Constraint-based synthesis of Datalog programs*. In Christopher Beck, editor: *23rd International Conference on Principles and Practice of Constraint Programming*, CP 2017, Springer, pp. 689–706.

[2] Lars Ole Andersen (1994): *Program analysis and specialization for the C programming language*. Ph.D. thesis, DIKU, University of Copenhagen.

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

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

[5] Pavol Bielik, Veselin Raychev & Martin Vechev (2017): *Learning a static analyzer from data*. In Rupak Majumdar & Viktor Kunčak, editors: *Computer Aided Verification*, Springer, pp. 233–253.

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

[7] Kwonsoo Chae, Hakjoo Oh, Kihong Heo & Hongseok Yang (2017): *Automatically generating features for learning program analysis heuristics for C-like languages*. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 101:1–101:25.

[8] James Cheney, Laura Chiticariu & Wang-Chiew Tan (2009): *Provenance in Databases: Why, How, and Where*. *Foundations and Trends in Databases* 1(4), pp. 379–474.

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

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

[11] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens & Luc De Raedt (2015): *Inference and learning in probabilistic logic programs using weighted boolean formulas*. *Theory and Practice of Logic Programming* 15(3), pp. 358–401, doi:10.1017/S1471068414000076.

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

[13] Lise Getoor & Ben Taskar, editors (2007): *Introduction to statistical relational learning*. Adaptive Computation and Machine Learning, MIT Press.

[14] Noah Goodman, Vikash Mansinghka, Daniel Roya, Keith Bonawitz & Joshua Tenenbaum (2008): *Church: A language for generative models*. In: *Proceedings of the 24th Conference Annual Conference on Uncertainty in Artificial Intelligence*, UAI 2008, AUAI Press, pp. 220–229.

[15] Todd Green, Grigoris Karvounarakis & Val Tannen (2007): *Provenance Semirings*. In: *Proceedings of the 26th ACM Symposium on Principles of Database Systems*, PODS 2007, ACM, pp. 31–40.

[16] Radu Grigore & Hongseok Yang (2016): *Abstraction refinement guided by a learnt probabilistic model*. In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages*, POPL 2016, ACM, pp. 485–498.

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

[18] Sehun Jeong, Minseok Jeon, Sungdeok Cha & Hakjoo Oh (2017): *Data-driven context-sensitivity for points-to analysis*. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 100:1–100:28.

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

[20] Omer Katz, Ran El-Yaniv & Eran Yahav (2016): *Estimating types in binaries using predictive modeling*. In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages*, POPL 2016, ACM, pp. 313–326.

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

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

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

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

[25] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh & Kwangkeun Yi (2017): *Sound Non-Statistical Clustering of Static Analysis Alarms*. *ACM Trans. Program. Lang. Syst.* 39(4), pp. 16:1–16:35, doi:10.1145/3095021. Available at `http://doi.acm.org/10.1145/3095021`.

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

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

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

[29] Stephen Muggleton & Luc de Raedt (1994): *Inductive logic programming: Theory and methods*. *The Journal of Logic Programming* 19–20, pp. 629–679. Special Issue: Ten Years of Logic Programming.

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

[31] Hakjoo Oh, Hongseok Yang & Kwangkeun Yi (2015): *Learning a strategy for adapting a program analysis via Bayesian optimisation*. In: *Proceedings of the 2015 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, ACM, pp. 572–588.

[32] Zvonimir Pavlinovic, Tim King & Thomas Wies (2014): *Finding minimum type error sources*. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2014, ACM, pp. 525–542.

[33] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo & Mayur Naik (2018): *User-guided program reasoning using Bayesian inference*. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (To appear)*, PLDI, ACM.

[34] Veselin Raychev, Martin Vechev & Andreas Krause (2015): *Predicting program properties from "Big Code"*. In: *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages*, POPL 2015, ACM, pp. 111–124.

[35] Thomas Reps (1995): *Demand interprocedural program analysis using logic databases*, pp. 163–196. Springer.

[36] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris & Mayur Naik (2018): *Syntax-guided synthesis of Datalog programs*. Technical Report, University of Pennsylvania. In submission.

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

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

[39] Xin Zhang, Radu Grigore, Xujie Si & Mayur Naik (2017): *Effective interactive resolution of static analysis alarms*. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 57:1–57:30.