EFFECTIVE PROGRAM REASONING USING BAYESIAN INFERENCE

Sulekha Kulkarni

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2020

Supervisor of Dissertation

Mayur Naik, Professor, Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor, Computer and Information Science

Dissertation Committee:

Rajeev Alur, Zisman Family Professor of Computer and Information Science

Val Tannen, Professor of Computer and Information Science

Osbert Bastani, Research Assistant Professor of Computer and Information Science

Suman Nath, Partner Research Manager, Microsoft Research, Redmond

To my father,

who set me on this path,

to my mother,

who leads by example,

and

to my husband,

who is an infinite source of courage.

Acknowledgments

I want to thank my advisor Prof. Mayur Naik for giving me the invaluable opportunity to learn and experiment with different ideas at my own pace. He supported me through the ups and downs of research, and helped me make the Ph.D. a reality.

I also want to thank Prof. Rajeev Alur, Prof. Val Tannen, Prof. Osbert Bastani, and Dr. Suman Nath for serving on my dissertation committee and for providing valuable feedback. I am deeply grateful to Prof. Alur and Prof. Tannen for their sound advice and support, and for going out of their way to help me through challenging times. I am also very grateful for Dr. Nath's able and inspiring mentorship during my internship at Microsoft Research, and during the collaboration that followed.

Dr. Aditya Nori helped me start my Ph.D. journey and was a source of encouragement throughout. During the early years of my graduate studies, Prof. Santosh Pande supported me with helpful advice. I thank them both. I also thank Prof. Sanjeev Khanna for his friendly encouragement during my time at Penn.

I thank all my collaborators, particularly Mukund Raghothaman, Ravi Mangal, Xin Zhang, and Kihong Heo. I learnt a lot from the stimulating and enjoyable discussions I had with them. I also thank all the other past and present members of our research group for their support: Woosuk Lee, Xujie Si, Anthony Canino, Hyonyoung Choi, Pardis Pashakhanloo, Halley Young, Elizabeth Dinella, Jiani Huang and Ziyang Li.

I thoroughly enjoyed the weekly lunch seminars, the colloquium talks, and interesting conversations with the vibrant PL community at Penn. I enjoyed going for walks and having friendly interactions with Farnaz Behrang, Arjun Radhakrishna, Kostas Mamouras, and Nimit Singhania. I also wish to thank all the staff members of the CIS department, RAS and ISSS whose patience and cheerful countenances made it possible for me to easily execute administrative tasks.

I want to thank my late father, Narasinh Kanekal, whose passion for physics and math laid the foundation for the path his two daughters would later pursue. I thank my mother, Pushpa Kanekal, who gave me her wholehearted support, encouraged me never to give up, and prayed for me. I am grateful to my daughters, Charu and Neeraja, who were always by my side on this journey. They helped me navigate cultural differences, gave me writing tips, and painstakingly reviewed my writing. I also thank my extended family for their encouragement and support throughout.

Most importantly, no words can express how much I owe my husband Raghu. He was a never-failing source of courage and helpful advice. This Ph.D. would not have been possible without his steadfast support.

ABSTRACT

EFFECTIVE PROGRAM REASONING USING BAYESIAN INFERENCE

Sulekha Kulkarni

Mayur Naik

Program analysis tools that statically find bugs in software still report a deluge of false alarms notwithstanding their widespread adoption. This is because they must necessarily make approximations in order to scale to large and complex programs. The focus of this dissertation is to make static program analyses more effective by guiding them towards true bugs and away from false alarms. We do this by augmenting logical program reasoning with probabilistic reasoning. We seek to overcome the incompleteness of a static analysis by associating each alarm it produces with a probability that it is a true alarm. We compute alarm probabilities by performing Bayesian inference on a probabilistic model derived from the execution of the analysis. Moreover, the probabilistic model allows us to recompute the probabilities by conditioning them on new evidence, thereby allowing to tailor the analysis to individual codebases and user needs. The alarms are ranked by the computed probabilities to mitigate the burden of inspecting false alarms.

We demonstrate the effectiveness of our approach in two practical systems. In one system, we leverage user feedback to iteratively improve the alarm ranking. The system starts with an initial ranking of alarms reported by the static analysis. In each iteration, the system seeks user feedback for the top-ranked alarm. Next, it generalizes this feedback by recomputing the probabilities of all the alarms conditioned on this feedback to produce an improved ranking for the next iteration. After a few iterations, true alarms rise to the top of the ranking, thus alleviating the burden of inspecting false alarms. In the second system, we leverage the completeness of dynamic analysis that is capable of observing concrete program executions, to rank the alarms reported by the static analysis. We hypothesize that a reported alarm is at most as complete as the analysis facts it is premised upon. For each analysis fact used by the static analysis in deducing an alarm, the system seeks a probability estimate for its completeness, from a dynamic analysis. The dynamic analysis estimates this by counting the number of times it observes the analysis fact during concrete program executions. The system then uses the estimated probabilities associated with the analysis facts to infer probabilities for alarms, and ranks alarms by the inferred probabilities.

Table of Contents

List of Tables

List of Illustrations

Chapter 1

Introduction

1.1 The Effectiveness of Program Reasoning

Program analysis tools, such as Facebook Infer [16], Clang Static Analyzer [2], Coverity Static Analyzer [9], and many others [12, 8, 69], have been developed over the last twenty years to statically find bugs in software. These tools rely on logical modes of reasoning about programs, such as abstract interpretation and symbolic execution, in order to find bugs. To be useful in realistic software development environments, such tools need to be sound, scalable, and precise. The soundness of a bug-finding program analysis tool ensures that no bug in the analyzed program is missed by the tool. The scalability of the tool measures the size of the program that it is able to reason about. The precision of the tool measures the fraction of the total reported bugs that are indeed true bugs. It is extremely difficult for a static program analysis tool to be simultaneously sound, scalable, as well as precise; optimizing for any one of these requirements compromises the others. In order to scale to large programs, a static program analysis tool must necessarily make approximations that may cause it to be imprecise and possibly unsound.

In practice, static program analysis tools strive to be effective by achieving an acceptable but delicate balance between the competing requirements of scalability, soundness, and precision. These tools achieve this balance by employing a mixture of sound and unsound approximations. In addition, the tools also provide configurable parameters that give tool users control over the tool's scalability, soundness, and precision. Tool users can fine-tune the values of the configurable parameters to optimize the tool's performance in specific software engineering environments. These configurations are, at best, *ad hoc* methods to improve the effectiveness of a static analysis tool. In sum, making static program analysis tools effective in practical environments remains a challenge.

1.2 The Problem of False Alarms

One of the major impediments to the effectiveness of static program analysis tools is the large number of false alarms they report. In other words, the static analyses employed by these tools have low precision. A user has to triage all the reported alarms in order to find the true alarms that are few and far apart. It is common for tool users to tune a tool's configurable parameters to report fewer false alarms in order to reduce the burden of inspecting these alarms, but parameter tuning is an *ad hoc* approach that may cause the tool to suppress alarms that are indeed critical bugs. For example, the Coverity Static Analyzer employs sound approximations that make it scalable, but these approximations cause the tool to report a large number of false alarms. The tool suppresses many of these false alarms using other approximations that are unsound. As a result, the Coverity Static Analyzer failed to detect a critical security bug famously known as the Heartbleed bug [3]. Similarly, the Clang Static Analyzer performs under-constrained symbolic execution in order to scale. While this approach is sound, it causes the analyzer to be imprecise and report a large number of false alarms. The tool provides certain configurable parameters to suppress false alarms, like parameters that govern when copy/move constructors need to be inlined for analysis, or when bug-paths that go through null returns need to be suppressed.

While such *ad hoc* methods may improve the tool's effectiveness in a given context, they may potentially suppress critical bugs. Therefore, we need systematic ways to distinguish true bugs from the false alarms.

In this dissertation, we focus on the problem of finding true bugs interspersed among numerous false alarms. We present a principled and general approach to rank the alarms reported by bug-finding static program analysis tools in order of their decreasing likelihood of being true alarms. By ranking alarms, we mitigate the burden of inspecting false alarms, thereby improving the effectiveness of such tools in practical settings. In our approach, we use the well-known fact that alarms are correlated: multiple true alarms often share root causes, and multiple false alarms are often caused by the inability of the analysis to prove some shared intermediate fact about the analyzed program. Indeed, a large body of previous research is aimed at alarm clustering [46, 48], ranking [42, 41], and classification [55, 81, 34].

1.3 A Probabilistic Approach to Ranking Alarms

We propose and implement a general approach to ranking the alarms reported by a static analysis by the alarms' probability of being true. Our approach augments the logical reasoning of a static analysis with probabilistic reasoning. This augmentation enables us to quantitatively model the incompleteness in static analyses, which is the primary source of false alarms. Our approach uses this model to infer a probability for each alarm that quantifies how likely the alarm is to be true. Next, our approach ranks alarms by their computed probabilities. Such a ranking has the potential to greatly improve the practical effectiveness of static program reasoning tools because it mitigates the burden of inspecting false alarms.

In our approach, we extract a probabilistic model of the execution of the static analysis and perform inference on that model. The probabilistic model is a Bayesian network, and it models both: (a) the analysis facts produced, and (b) the deduction steps executed by the static analysis while analyzing a program. In addition, the probabilistic model captures how each deduced analysis fact is conditionally dependent on its premises. Performing Bayesian inference on this model gives us a way to associate probabilities with all the analysis facts when we know the probabilities of certain analysis facts like input facts, or when we are given zero or more observations. Because an alarm is an analysis fact deduced by the static analysis, we can now associate an alarm with a probability. Furthermore, our probabilistic model allows us to recompute all the probabilities conditioned on new evidence that may be observed.

In this dissertation, we work with static program analyses that are specified as logical rules in Datalog, although our approach is applicable to all static program analyses that rely on reasoning techniques such as abstract interpretation. Datalog is a logic programming language widely used to declaratively specify complex program analyses [73, 79, 15, 6, 54, 5]. From the deductive steps applied by such analyses, we observe that correlated alarms share significant portions of their derivation trees (the sequence of analysis steps executed by the static analysis in deriving the alarm). The union of all instantiated analysis steps occurring in these derivation trees induces a derivation graph. The derivation graph serves as our starting point to capturing the (transitive) dependence of an analysis fact on its premises and, as a consequence, it enables us to capture alarm correlations as well.

We extract a Bayesian network from this derivation graph by representing each instantiated deduction rule (i.e., a clause) and each analysis fact (i.e., a tuple) as a node in the Bayesian network. The edges of the network represent conditional dependencies. A clause node is conditionally dependent on its "antecedent" tuple nodes, and a tuple node is conditionally dependent on the clause nodes that "derive" it. Every node is associated with a conditional probability distribution that quantifies the incompleteness of the clause or tuple that the node represents. Since a Bayesian network is acyclic by definition, such a straightforward manner of extracting a Bayesian network from a derivation graph is possible only if the derivation graph is acyclic. Therefore, we eliminate directed cycles from a derivation graph by deleting clauses to break cycles, despite the fact that we may introduce more incompleteness because of the loss of some derivation trees. But, cycle elimination is necessary for us to get a handle on the problem.

In a Bayesian network constructed in this manner, alarm probabilities are the posterior probabilities of the nodes representing the alarm tuples. Therefore, computing alarm probabilities reduces to performing marginal inference on the Bayesian network. If the evidence of the ground truth of some alarms becomes available in the future, we can recompute the posterior probabilities of the remaining alarms conditioned on this evidence. With this approach, we obtain a probability for each reported alarm that measures how likely it is that the alarm represents a real bug. We then rank alarms by their inferred probabilities, in decreasing order. This ranking places likely true alarms at the top, thereby enabling users to prioritize the triaging of likely true alarms over false ones.

Our approach performs approximate marginal inference, using the loopy belief propagation algorithm. We choose approximate inference because exact inference is infeasible on large Bayesian networks that result from static analyses applied to large and complex practical programs. In fact, the large sizes of the Bayesian networks in practical settings pose a scalability problem even to approximate inference algorithms. For this reason, our approach employs optimizations to reduce the size of a derivation graph. These optimizations cause a corresponding reduction in the size of the Bayesian network.

While our approach specifies a general way to augment program reasoning with probabilistic reasoning, a specific instantiation of our approach needs to choose the parameters of the Bayesian network. An instantiation of our approach needs to:

1. Define the conditional probability distribution (CPD) table at each node of the Bayesian network. The CPD table at a node models the incompleteness of the clause or tuple represented by that node.

2. Choose a cycle elimination algorithm and applicable optimizations. These choices are made with a view to increasing the empirical effectiveness of alarm ranking.

1.3.1 End-to-End Systems

We demonstrate the effectiveness and the generality of our approach by realizing it in two practical end-to-end systems called BINGO [70] and PRESTO. Both BINGO and PRESTO instantiate our general approach in different ways by making different choices with respect to the following:

1. Defining the CPD tables of the Bayesian network in order to capture the sources of incompleteness of the specific static analyses executed by the system.

2. Choosing the cycle elimination algorithm, and the optimizations applied to the derivation graph to achieve empirical effectiveness of alarm ranking.

BINGO employs aggressive cycle elimination and two aggressive optimizations. BINGO chooses aggressive cycle elimination in order to prune away large parts of the extremely large derivation graphs produced by the static analyses executed by BINGO, namely the datarace analysis and the taint analysis. One of the reasons these analyses produce extremely large derivation graphs is because they perform deductions at the granularity of individual program instructions. In addition, BINGO employs aggressive optimizations to further reduce the size of the derivation graph.

BINGO defines the CPD tables associated with clause nodes to be probabilistic in order to capture our hypothesis that incomplete analysis rules cause the datarace and taint analyses to be incomplete. That is, for a clause node, BINGO specifies that its consequent holds true with a probability that is strictly less than 1, even when all its antecedents hold true. In addition, BINGO treats the analysis input facts as having a probability of 1. The intuition underlying this hypothesis stems from the fact that the datarace analysis is path-insensitive and the taint analysis is flow-insensitive.

Furthermore, Bingo leverages user feedback to compute and iteratively improve the alarm ranking. It first computes an initial ranking of alarms. In an interactive loop involving a user, Bingo seeks feedback about the ground truth of the top-ranked alarm. Next, it re-ranks alarms by recomputing the probabilities of all the alarms conditioned on this feedback. In effect, this recomputation generalizes user feedback, bringing correlated alarms closer to each other in the new ranking. Such user interactions are repeated iteratively, causing true alarms to rise to the top of the ranking. This process of iteratively improving the alarm ranking is a key factor in the effectiveness of Bingo.

Presto employs a depth-first-search-based cycle elimination algorithm that retains more derivation trees than the aggressive cycle elimination algorithm. Further, Presto employs only one aggressive size-reducing optimization on the derivation graph. Presto can afford to be less aggressive while eliminating cycles from the derivation graph as compared to Bingo because the derivation graphs produced by the static analysis executed by Presto, namely the exception flow analysis, are much smaller as compared to, say, datarace derivation graphs. One of the reasons is because the exception flow analysis, operates at the granularity of methods (performs deductions on the nodes and edges of a call-graph). As a result of being less aggressive during cycle elimination, Presto retains most of the derivation trees in a derivation graph.

Presto defines the CPD tables associated with input tuples to be probabilistic, and treats the deductive rules of the analysis as complete, in order to capture our hypothesis that incomplete input tuples are the primary sources of false alarms in the case of the exception flow analysis. A bug-finding static analysis, like the exception flow analysis, is typically built atop one or more underlying analyses that produce intermediate facts such as the call-graph and aliasing information. These intermediate facts, which serve as inputs to the bug-finding analysis, are sometimes incomplete because they are produced by analyses that make approximations. Therefore, Presto models the incompleteness of such input facts, in the CPD tables associated with them, by specifying that an input fact holds true with a probability that is strictly less than one.

In addition, Presto leverages the completeness of dynamic analysis to rank alarms reported by a static analysis. Presto seeks to quantify the incompleteness of each input fact by a probability estimate: it seeks this estimate from a dynamic analysis that computes it by observing concrete program executions. Next, Presto constructs a Bayesian network as described earlier. Marginal inference on the Bayesian network propagates these probability estimates, and infers the posterior probabilities associated with all the nodes of the network. Presto then ranks alarms by their inferred probabilities.

1.3.2 Cycle Elimination

Finally, we examine the problem of cycle elimination, which arises in the context of extracting a Bayesian network from a derivation graph. Because a Bayesian network is acyclic by definition, we need to remove directed cycles from a derivation graph in order to convert it into a Bayesian network in a straightforward manner. Eliminating cycles in a derivation graph introduces more incompleteness into it because we may lose some derivations of some analysis facts. In this dissertation, we present three algorithms for cycle elimination, each of which is more precise than the previous algorithm in the number of derivation trees it retains for each analysis fact.

1.4 Contributions and Organization

In summary, our contributions in this dissertation are as follows:

1. A general approach to augmenting program reasoning with probabilistic reasoning.

2. Two end-to-end systems that realize and illustrate our approach.

3. A study of three approaches to cycle elimination in derivation graphs, a key step in extracting a probabilistic model from the derivation steps of a static analysis.

The rest of the material in this dissertation is organized as follows. Chapter 2 covers background material, and Chapter 3 describes our approach to extending program reasoning with probabilistic reasoning. Chapter 4 describes the two end-to-end systems, Bingo and Presto, that rank alarms produced by client static analyses: a datarace analysis, a taint analysis, and an exception flow analysis. Chapter 5 explores different approaches to cycle elimination. Chapter 6 discusses related work, Chapter 7 suggests directions in which this research can be continued, and finally, Chapter 8 concludes.

Chapter 2

Background

In this chapter, we cover the necessary background material on Datalog (Section 2.1) and on Bayesian networks (Section 2.2).

2.1 Datalog Programs

In this section, we introduce the syntax and semantics of Datalog programs. A more detailed treatment is in [4]. A Datalog program comprises a set of rules or constraints of the form

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

Here, each $R_i(\boldsymbol{u}_i)$ is a *relation* representing a set of tuples with name R_i. The relation $R_i(\boldsymbol{u}_i)$ has the form $R_i(u_1, u_2, \ldots, u_k)$ and arity k where $u_1, u_2, \ldots, u_k$ are free variables. The tuple $R_i(\boldsymbol{v}_i) = R_i(v_1, v_2, \ldots, v_k)$ instantiates relation $R_i(\boldsymbol{u}_i)$ with atoms $v_1, v_2, \ldots, v_k$ drawn from appropriate domains whose union is the active domain of constants. The set of all tuples $R_i(\boldsymbol{v}_i)$ is the *extent* of the relation $R_i(\boldsymbol{u}_i)$. A relation whose extent is completely specified to us is called a relation of the extensional database, or an EDB relation.

Each rule may be read as a universally quantified formula: "For all instantiations $\boldsymbol{v}$ of the free variables from the active domain of constants, if $R_1(\boldsymbol{v}_1)$, and $R_2(\boldsymbol{v}_2)$,

$\ldots$, and $R_p(\boldsymbol{v}_p)$, then $R_h(\boldsymbol{v}_h)$". Instantiating a Datalog rule yields the Horn clause $R_1(\boldsymbol{v}_1) \wedge R_2(\boldsymbol{v}_2) \wedge \cdots \wedge R_p(\boldsymbol{v}_p) \implies R_h(\boldsymbol{v}_h)$. An instantiated rule is also referred to as a grounded constraint, or as a clause. Only one relation can occur to the left of the :– sign in a Datalog rule and this relation is called the *head* or *goal* of the rule. Zero or more relations can occur to the right of the :– sign and they form the *body* or *subgoals* of the rule. Thus a rule specifies how to compute tuples of the head relation from the tuples of relations in the body. The relations whose extents are computed by Datalog rules are called the relations of the intensional database, or IDB relations. The EDB (IDB) relations along with their extents form the EDB (IDB). One or more of the IDB relations are designated to be the *output* relations. A Datalog program, is therefore, a function from the EDB to the IDB.

Solving a Datalog program, entails the following actions. Given the extent I of all EDB relations, we initialize the set of tuples, $T \coloneqq I$, and initialize the grounded constraints to the set of constraints that derive the input tuples, $GC \coloneqq \{\text{True} \implies t \mid t \in I\}$. We repeatedly apply each rule to update T and GC until no new tuple is derived. That is, whenever $R_1(\boldsymbol{v}_1)$, $R_2(\boldsymbol{v}_2)$, $\ldots$, $R_p(\boldsymbol{v}_p)$ occur in T, we update $T \coloneqq T \cup \{R_h(\boldsymbol{v}_h)\}$, and $GC \coloneqq GC \cup \{R_1(\boldsymbol{v}_1) \wedge R_2(\boldsymbol{v}_2) \wedge \cdots \wedge R_p(\boldsymbol{v}_p) \implies R_h(\boldsymbol{v}_h)\}$. The set of tuples T is said to be the solution to the Datalog program. This solution is said to have reached a fixpoint when the Datalog program applied to T, yields T. A Datalog program is a monotone function that has a least fixpoint.

2.2 Bayesian Networks

We define and briefly explain a Bayesian network. A more detailed treatment is in [40]. Let V be a set of random variables. For our purpose, it suffices to consider only boolean-valued random variables. Let $\mathcal{G} = (V, E)$ be a directed acyclic graph with its vertices as the set of random variables V. Let E be a set of directed edges. Edges represent conditional dependencies. If there is no directed path from one node to another, then those two nodes represent random variables that are conditionally independent of one another. Each random variable is associated with a probability function called a *conditional probability distribution* (CPD) that is defined below. For any $v \in V$, let $\mathrm{Pa}(v)$ be the set of variables with edges leading to v. Formally, $\mathrm{Pa}(v) = \{u \in V \mid u \to v \in E\}$. The CPD of a random variable v is a function that maps concrete valuations $\boldsymbol{x}_{\mathrm{Pa}(v)}$ of $\mathrm{Pa}(v)$ to the conditional probability of the event $v = \mathrm{True}$, and we write this as $p(v \mid \boldsymbol{x}_{\mathrm{Pa}(v)})$. The complementary event $v = \mathrm{False}$ has conditional probability $p(\neg v \mid \boldsymbol{x}_{\mathrm{Pa}(v)}) = 1 - p(v \mid \boldsymbol{x}_{\mathrm{Pa}(v)})$. The Bayesian network, given by the triple $BN = (V, \mathcal{G}, p)$, is a compact representation of the following joint probability distribution:

$$\Pr(\boldsymbol{x}) = \prod_{v} p(x_v \mid \boldsymbol{x}_{\mathrm{Pa}(v)}), \tag{2.1}$$

where the joint assignment $\boldsymbol{x}$ is a valuation x_v for each $v \in V$, and $\boldsymbol{x}_{\mathrm{Pa}(v)}$ is the valuation restricted to the parents of v. That is, a Bayesian network represents a family of joint distributions that factorize as a product of *local* conditional probabilities. From Equation 2.1, the following definitions can be derived: (a) the marginal probability of a variable, $\Pr(v) = \sum_{\{\boldsymbol{x} \mid x_v = \mathrm{True}\}} \Pr(\boldsymbol{x})$, $\Pr(\neg v) = 1 - \Pr(v)$, and (b) the conditional probability of arbitrary events: $\Pr(v \mid \boldsymbol{e}) = \Pr(v \wedge \boldsymbol{e}) / \Pr(\boldsymbol{e})$.

Chapter 3

Augmenting Program Reasoning with Probabilistic Reasoning

In this dissertation, we work with static analyses that make deductive steps explicit by specifying them in Datalog. But our approach is applicable to all static analyses that employ techniques like abstract interpretation. This is because all such static analyses apply, in effect, a series of deductive steps: at every program point, they draw a conclusion based on the premises that hold at that program point.

The set of all deductive steps that have been executed by a static analysis specified in Datalog, is captured as a Datalog derivation graph. We first describe this process in Section 3.1. Then we go on to describe how a Bayesian network is constructed from a derivation graph (Section 3.2). Next, we discuss marginal inference on a Bayesian network (Section 3.3), and illustrate how alarm probabilities are conditioned on evidence (Section 3.4). We end this section by discussing the configurable parameters of our Bayesian network (Section 3.5).

3.1 Static Analysis to Derivation Graph

In the scope of our study, a static analysis is a Datalog program. When a static analysis specified in Datalog is applied to a program, the Datalog solver instantiates the rules of the static analysis using EDB facts (i.e., tuples) that are extracted from the program text. These rule instantiations produce new IDB facts (i.e., tuples) that are used by the Datalog solver to instantiate more rules; this process continues until no more new IDB facts are produced. At this point, we say that a fixpoint is reached.

3.1.1 Datalog Derivation Graph

The Datalog derivation graph $\mathcal{G}(T, GC)$ is induced by the set of tuples T and the grounded constraints GC from the solution to a Datalog program, at fixpoint. The set of vertices is $T \cup GC$. There is an edge from a tuple $t \in T$ to a clause $g \in GC$ whenever t is an antecedent of g, and an edge from g to t whenever t is the consequent of g. We explain the structure of a derivation graph with an example. We show a snippet of Java code in Figure 3.1, and the execution of a highly simplified static datarace analysis specified in Figure 3.2, on the code snippet.

Consider the Java code in Figure 3.1. Assume that the methods `close()` and `getRequest()` may be simultaneously invoked by multiple threads on the same object. The `synchronized` block on lines `L1`–`L3` ensures that for each object instance, lines `L4`–`L7` are executed at most once. Therefore there is no datarace between the pair of accesses to `controlSocket` on lines `L4` and `L5`, and between accesses on lines `L5` and `L5`. There is also no datarace between the accesses to `request` on lines `L6` and `L7`, and between accesses on lines `L7` and `L7` by different threads. However, there is a potential datarace between accesses to `request` on lines `L0` and `L7`.

Figure 3.2 shows an extremely simplified static datarace analysis in Datalog for Java programs, minimized from the one present in the Chord program analysis framework [64]. The analysis takes the relations $\mathrm{N}(p_1, p_2)$, $\mathrm{U}(p_1, p_2)$, and $\mathrm{A}(p_1, p_2)$ as

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

   ...

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

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

Figure 3.1: Example code fragment in Java.

input, and produces the relations $\mathrm{P}(p_1, p_2)$ and $\mathrm{race}(p_1, p_2)$ as output. In all these relations, variables p_1, p_2 and p_3 range over the domain of program points. The input relations contain tuples indicating some known facts about the program. For example, for the program in Figure 3.1, $\mathrm{N}(p_1, p_2)$ may contain the tuples $\mathrm{N}(\mathtt{L1}, \mathtt{L2})$, $\mathrm{N}(\mathtt{L2}, \mathtt{L3})$, etc. Some input relations such as $\mathrm{N}(p_1, p_2)$ are directly obtained by analyzing the program text. Other input relations such as $\mathrm{U}(p_1, p_2)$ or $\mathrm{A}(p_1, p_2)$ are outputs of earlier analyses (in this case, a lockset analysis and a pointer analysis, respectively). The Datalog solver applies the analysis rules described in Figure 3.2, to the input tuples until fixpoint.

The analysis rules should be understood as follows:

1. Rule r_1 : For all program points p_1, p_2, p_3, if p_1 and p_2 may execute in parallel ($\mathrm{P}(p_1, p_2)$), and p_3 may execute immediately after p_2 ($\mathrm{N}(p_2, p_3)$), and p_1 and p_3 are not guarded by a common lock ($\mathrm{U}(p_1, p_3)$), then p_1 and p_3 may themselves execute in parallel.

2. Rule r_2 : For all program points p_1, p_2, if p_1 and p_2 may execute in parallel

Input relations	
$\mathrm{N}(p_1, p_2)$:	Program point p_2 may be executed immediately after program point p_1 by a thread.
$\mathrm{U}(p_1, p_2)$:	Program points p_1 and p_2 are not guarded by a common lock.
$\mathrm{A}(p_1, p_2)$:	Program points p_1 and p_2 may access the same memory location.
Output relations	
$\mathrm{P}(p_1, p_2)$:	Program points p_1 and p_2 may be executed by different threads in parallel.
$\mathrm{race}(p_1, p_2)$:	Program points p_1 and p_2 may have a datarace.
Analysis rules	

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

Figure 3.2: A toy static datarace analysis in Datalog. The base rule for relation $\mathrm{P}(p_1, p_2)$ is elided for brevity.

$(\mathrm{P}(p_1, p_2))$, then p_2 and p_1 may execute in parallel.

3. Rule r_3 : For all program points p_1, p_2, if p_1 and p_2 may execute in parallel $(\mathrm{P}(p_1, p_2))$, and p_1 and p_2 may access the same memory location $(\mathrm{A}(p_1, p_2))$, then p_1 and p_2 may have a datarace.

A rule fires if and only if all its antecedent tuples are among the input tuples or have been derived. On the other hand, a (consequent) tuple may be derived by more that one rule. All instantiations of these rules together induce the derivation graph.

Figure 3.3 shows a snippet of the derivation graph when the rules above are applied to the program snippet in Figure 3.1. A clause node is interpreted as conjunctive: a clause is considered to have fired if and only if all its antecedent tuples have been derived. Whereas, a tuple node is interpreted as disjunctive: a tuple is considered derived if there is at least one clause deriving it. Also note that the two “alarm” nodes race(`L4`, `L5`) and race(`L6`, `L7`) are correlated because they (transitively) depend on the same analysis fact P(`L4`, `L5`). That is, the two alarms share the portion of the derivation tree that derives fact P(`L4`, `L5`).

$$\begin{tikzcd}
& \boxed{\mathrm{P}(\mathrm{L4},\mathrm{L2})} \arrow[d] & \boxed{\mathrm{N}(\mathrm{L2},\mathrm{L3})} \arrow[dl] & \boxed{\mathrm{U}(\mathrm{L4},\mathrm{L3})} \arrow[dll] \\
& r_1(\mathrm{L4},\mathrm{L2},\mathrm{L3}) \arrow[d] & & \\
& \boxed{\mathrm{P}(\mathrm{L4},\mathrm{L3})} \arrow[d] & \boxed{\mathrm{N}(\mathrm{L3},\mathrm{L4})} \arrow[dl] & \boxed{\mathrm{U}(\mathrm{L4},\mathrm{L4})} \arrow[dll] \\
& r_1(\mathrm{L4},\mathrm{L3},\mathrm{L4}) \arrow[d] & & \\
& \boxed{\mathrm{P}(\mathrm{L4},\mathrm{L4})} \arrow[d] & \boxed{\mathrm{N}(\mathrm{L4},\mathrm{L5})} \arrow[dl] & \boxed{\mathrm{U}(\mathrm{L4},\mathrm{L5})} \arrow[dll] \\
& r_1(\mathrm{L4},\mathrm{L4},\mathrm{L5}) \arrow[d] & & \\
\boxed{\mathrm{A}(\mathrm{L4},\mathrm{L5})} \arrow[dr] & \boxed{\mathrm{P}(\mathrm{L4},\mathrm{L5})} \arrow[d] \arrow[drr] & \boxed{\mathrm{N}(\mathrm{L5},\mathrm{L6})} \arrow[dr] & \boxed{\mathrm{U}(\mathrm{L4},\mathrm{L6})} \arrow[d] \\
& r_3(\mathrm{L4},\mathrm{L5}) \arrow[d] & & r_1(\mathrm{L4},\mathrm{L5},\mathrm{L6}) \arrow[d] \\
& \boxed{\boxed{\mathrm{race}(\mathrm{L4},\mathrm{L5})}} & & \boxed{\mathrm{P}(\mathrm{L4},\mathrm{L6})} \arrow[d] \\
& & & r_2(\mathrm{L4},\mathrm{L6}) \arrow[d] \\
& \boxed{\mathrm{U}(\mathrm{L6},\mathrm{L5})} \arrow[drr] & \boxed{\mathrm{N}(\mathrm{L4},\mathrm{L5})} \arrow[dr] & \boxed{\mathrm{P}(\mathrm{L6},\mathrm{L4})} \arrow[d] \\
& & & r_1(\mathrm{L6},\mathrm{L4},\mathrm{L5}) \arrow[d] \\
& \boxed{\mathrm{U}(\mathrm{L6},\mathrm{L6})} \arrow[drr] & \boxed{\mathrm{N}(\mathrm{L5},\mathrm{L6})} \arrow[dr] & \boxed{\mathrm{P}(\mathrm{L6},\mathrm{L5})} \arrow[d] \\
& & & r_1(L6,\mathrm{L5},\mathrm{L6}) \arrow[d] \\
& \boxed{\mathrm{U}(\mathrm{L6},\mathrm{L7})} \arrow[drr] & \boxed{\mathrm{N}(\mathrm{L6},\mathrm{L7})} \arrow[dr] & \boxed{\mathrm{P}(\mathrm{L6},\mathrm{L6})} \arrow[d] \\
& & & r_1(\mathrm{L6},\mathrm{L6},\mathrm{L7}) \arrow[d] \\
& & \boxed{\mathrm{A}(\mathrm{L6},\mathrm{L7})} \arrow[dr] & \boxed{\mathrm{P}(\mathrm{L6},\mathrm{L7})} \arrow[d] \\
& & & r_3(\mathrm{L6},\mathrm{L7}) \arrow[d] \\
& & & \boxed{\boxed{\mathrm{race}(\mathrm{L6},\mathrm{L7})}}
\end{tikzcd}$$

Figure 3.3: A derivation graph snippet obtained by applying the datarace analysis in Figure 3.2 to the program in Figure 3.1.

3.1.2 Extracting the Derivation Graph

We extract the derivation graph by capturing all the rule instantiations at fixpoint. To do this, we produce an *instrumented* Datalog program that contains the same Datalog rules as the original program except that each rule is instrumented. The goal of instrumenting each rule is to record all the variables that get projected out while executing the rule. Executing the instrumented Datalog program will allow us to reconstruct all rule instantiations at fixpoint. We illustrate this process with an example rule. Consider rule r_1. When this rule executes to produce tuple $\mathrm{P}(p_1, p_3)$, the variable p_2 is projected out. We record all the variables that have been projected out by instrumenting the rules as follows:

$$\begin{aligned}
ir_1: &\quad \mathrm{P\_i1}(p_1, p_3, p_2) \coloneq \mathrm{P}(p_1, p_2), \mathrm{N}(p_2, p_3), \mathrm{U}(p_1, p_3).\\
ir_2: &\quad \mathrm{P\_i2}(p_2, p_1) \coloneq \mathrm{P}(p_1, p_2).\\
ir_3: &\quad \mathrm{race\_i1}(p_1, p_2) \coloneq \mathrm{P}(p_1, p_2), \mathrm{A}(p_1, p_2).
\end{aligned}$$

When the instrumented analysis is executed, every tuple from the instrumented relations $\mathrm{P\_i1}(p_1, p_3, p_2)$, $\mathrm{P\_i2}(p_2, p_1)$ and $\mathrm{race\_i1}(p_1, p_2)$ will give us the values of the constants from which we can reconstruct a grounded constraint.

For example, assume we have the input tuples $\mathrm{P}(\mathtt{L6}, \mathtt{L6})$, $\mathrm{N}(\mathtt{L6}, \mathtt{L7})$, $\mathrm{U}(\mathtt{L6}, \mathtt{L7})$ and $\mathrm{P}(\mathtt{L7}, \mathtt{L6})$. When the instrumented analysis executes, rule ir_1 will fire and produce tuple $\mathrm{P\_i1}(\mathtt{L6}, \mathtt{L7}, \mathtt{L6})$. From this tuple, we can reconstruct the fact that rule r_1 had fired in the original analysis. Similarly, rule ir_2 of the instrumented analysis will also fire producing tuple $\mathrm{P\_i2}(\mathtt{L6}, \mathtt{L6})$. The presence of this tuple will tell us that rule r_2 of the original analysis had fired. Executing the instrumented analysis after the original analysis has executed, will help extract the set of grounded constraints produced by the execution of the original analysis. From this set of grounded constraints, we can construct the derivation graph.

3.2 Derivation Graph to Bayesian Network

The next step is to construct a Bayesian network from the derivation graph. We have seen that a directed acyclic graph underlies a Bayesian network. The derivation graph is a directed graph but it may contain cycles. The derivation graph needs to be acyclic in order to extract a Bayesian network from it. Section 3.2.1 discusses cycle elimination in derivation graphs, and Section 3.2.2 describes the construction of the Bayesian network from the derivation graph. Finally, Section 3.2.3 discusses two optimizations performed on the derivation graph in order to reduce the size of the Bayesian network.

3.2.1 Cycle Elimination in Derivation Graphs

Figure 3.4 shows an example of a directed cycle in a derivation graph. We wish to extract a Bayesian network from a derivation graph by representing every node and edge in the derivation graph by a corresponding node and edge in the Bayesian network. A Bayesian network is by definition acyclic. Therefore, in order to convert a derivation graph into a Bayesian network in such a straightforward manner, we need to eliminate directed cycles from the derivation graph. Cycle elimination will introduce more incompleteness in the derivation of analysis facts, but it is a necessary step for generating our probabilistic model.

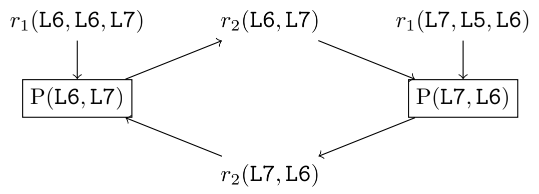


Figure 3.4: An example of a cycle in the derivation graph.

The process of cycle elimination chooses a subset of clauses GC_c from the set of all clauses GC, such that GC_c induces an acyclic derivation graph. Every tuple that is derivable in GC should still be derivable in GC_c. Moreover, we want to retain as many correlations between alarms (shared part of their derivation trees) as possible. Finding the largest such GC_c is an NP-complete problem (by reduction from the *maximum acyclic subgraph* problem [27]). We therefore relax the condition that GC_c must be the largest possible, and explore different approaches to finding an acyclic subset of clauses $GC_c \subseteq GC$. This is described in detail in Chapter 5 where we propose three algorithms for cycle elimination, which differ in the number of derivation trees they retain for tuples that are derived.

3.2.2 Acyclic Derivation Graph to Bayesian Network

We now convert GC_c, the acyclic subset of all clauses, to a Bayesian network, by following the guidelines below:

1. We represent each tuple and each clause by a node in the Bayesian network. For every edge in the derivation graph, there is a corresponding edge in the Bayesian network.

2. We quantify the incompleteness of each clause with a probability that represents the belief that the clause has an invalid conclusion despite having valid hypotheses.

3. We quantify the incompleteness of each input tuple with a probability that represents the likelihood of an input tuple not holding true.

4. We treat clause nodes as conjunctive nodes: that is, a clause could fire only if all its antecedent tuples are derived. We treat tuple nodes as disjunctive nodes: that is, a tuple could be derived by one or more clauses.

We apply these guidelines to an example clause and tuples. Rule r_1 in Figure 3.2 is incomplete. To see this, consider the following example. Even though the analysis facts P(`L4`, `L2`), N(`L2`, `L3`) and U(`L4`, `L3`) are all true, we know that fact P(`L4`, `L3`) is not true because if one thread executes line `L4`, other threads will return at line `L2` as explained earlier. We quantify the incompleteness of rules by associating a probability with clause nodes as follows:

$$\Pr(r_1(\texttt{L4}, \texttt{L2}, \texttt{L3}) \mid h_1) = 0.95, \text{ and} \tag{3.1}$$

$$\Pr(\neg r_1(\texttt{L4}, \texttt{L2}, \texttt{L3}) \mid h_1) = 1 - 0.95 = 0.05, \tag{3.2}$$

where $h_1 = \mathrm{P}(\texttt{L4}, \texttt{L2}) \wedge \mathrm{N}(\texttt{L2}, \texttt{L3}) \wedge \mathrm{U}(\texttt{L4}, \texttt{L3})$ is the event indicating that all the hypotheses of $r_1(\texttt{L4}, \texttt{L2}, \texttt{L3})$ are true, and 0.95 is the probability of the clause "firing". By setting the probability to a value strictly less than 1, we make it possible for the clause $r_1(\texttt{L4}, \texttt{L2}, \texttt{L3})$ to have not "fired", even though all the hypotheses indicated by event h_1 are true.

If any of the antecedents of $r_1(\texttt{L4}, \texttt{L2}, \texttt{L3})$ is false, then it is itself definitely false:

$$\Pr(r_1(\texttt{L4}, \texttt{L2}, \texttt{L3}) \mid \neg h_1) = 0, \text{ and} \tag{3.3}$$

$$\Pr(\neg r_1(\texttt{L4}, \texttt{L2}, \texttt{L3}) \mid \neg h_1) = 1. \tag{3.4}$$

Input tuples can also be incomplete because input tuples like $\mathrm{A}(\texttt{L6}, \texttt{L7})$ are produced by other program analyses that are themselves incomplete. Therefore, we associate a probability with input nodes as follows:

$$\Pr(\mathrm{A}(\texttt{L6}, \texttt{L7})) = 0.95, \text{ and} \tag{3.5}$$

$$\Pr(\neg \mathrm{A}(\texttt{L6}, \texttt{L7})) = 1 - 0.95 = 0.05. \tag{3.6}$$

Certain input tuples, for example $\mathrm{N}(\texttt{L6}, \texttt{L7})$, that are extracted directly from the program text, may be treated as known with certainty. In this case, the probability we associate with such input nodes is as follows:

$$\Pr(\mathrm{N}(\texttt{L6}, \texttt{L7})) = 1.0, \text{ and} \tag{3.7}$$

$$\Pr(\neg \mathrm{N}(\texttt{L6}, \texttt{L7})) = 0.0. \tag{3.8}$$

We treat nodes representing derived tuples as disjunctions:

$$\Pr(\mathrm{P}(\texttt{L6},\texttt{L7}) \mid r_1(\texttt{L6},\texttt{L6},\texttt{L7}) \vee r_2(\texttt{L7},\texttt{L6})) = 1, \text{ and} \tag{3.9}$$

$$\Pr(\mathrm{P}(\texttt{L6},\texttt{L7}) \mid \neg(r_1(\texttt{L6},\texttt{L6},\texttt{L7}) \vee r_2(\texttt{L7},\texttt{L6}))) = 0. \tag{3.10}$$

The probability value of 0.95 that we have used above is only for illustration. In general, the Bayesian network is parameterized by a vector of probabilities $\boldsymbol{p}$ that maps (a) each rule r to its rule firing probability that quantifies completeness of the rule, and/or (b) each instantiated input tuple t to a probability that quantifies its completeness. We discuss how to get the initial rule probabilities in Section 4.1.1. For an EDB relation that is treated probabilistically, Section 4.3 illustrates one approach to get the probabilities for individual tuples of the EDB relation. We associate conditional probability distributions (CPDs) with all nodes of the Bayesian network that has been constructed from the derivation graph induced by GC_c. Figure 3.5 shows a small snippet of the derivation graph that contains EDB tuple nodes, clause nodes that are interpreted as conjunctive nodes, and IDB tuple nodes that are interpreted as disjunctive nodes. Figure 3.6 shows the Bayesian network that corresponds to this snippet, with CPDs for representative nodes. Note that the size of the CPD for a node is exponential in the number of nodes on which it is conditionally dependent.

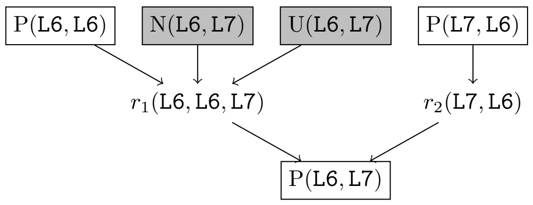
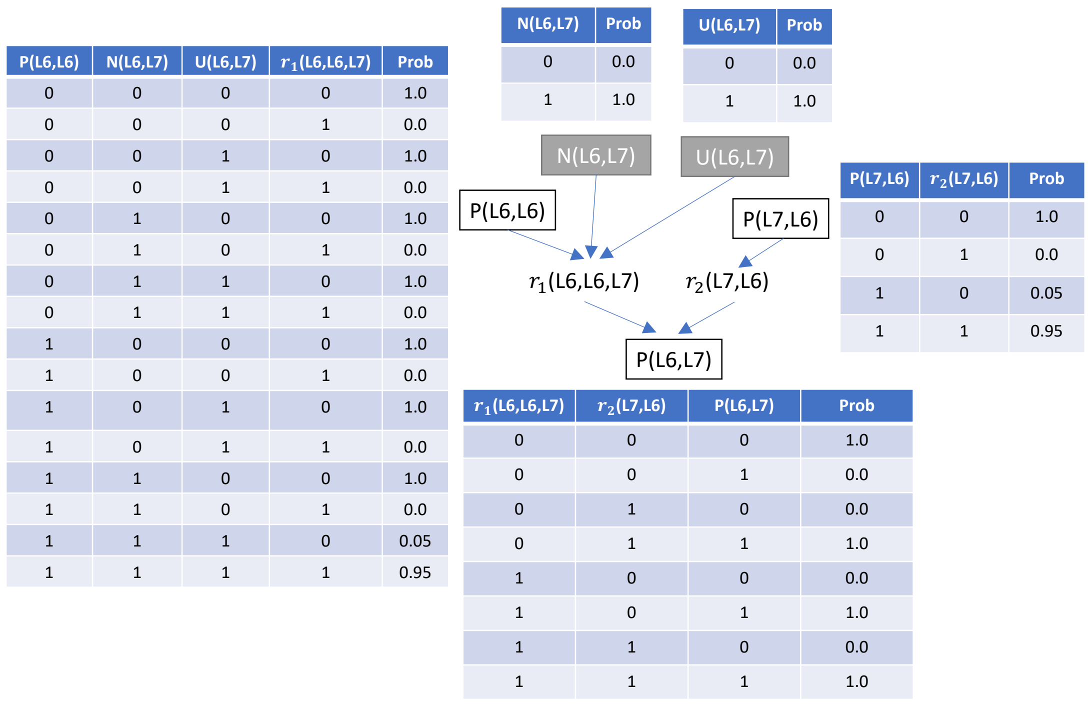


Figure 3.5: A derivation graph snippet showing a conjunctive node ($r_1(\texttt{L6},\texttt{L6},\texttt{L7})$) and a disjunctive node ($\mathrm{P}(\texttt{L6},\texttt{L7})$).



N(L6,L7)	Prob
0	0.0
1	1.0

U(L6,L7)	Prob
0	0.0
1	1.0

P(L6,L6)	N(L6,L7)	U(L6,L7)	r_1(L6,L6,L7)	Prob
0	0	0	0	1.0
0	0	0	1	0.0
0	0	1	0	1.0
0	0	1	1	0.0
0	1	0	0	1.0
0	1	0	1	0.0
0	1	1	0	1.0
0	1	1	1	0.0
1	0	0	0	1.0
1	0	0	1	0.0
1	0	1	0	1.0
1	0	1	1	0.0
1	1	0	0	1.0
1	1	0	1	0.0
1	1	1	0	0.05
1	1	1	1	0.95

P(L7,L6)	r_2(L7,L6)	Prob
0	0	1.0
0	1	0.0
1	0	0.05
1	1	0.95

r_1(L6,L6,L7)	r_2(L7,L6)	P(L6,L7)	Prob
0	0	0	1.0
0	0	1	0.0
0	1	0	0.0
0	1	1	1.0
1	0	0	0.0
1	0	1	1.0
1	1	0	0.0
1	1	1	1.0

Figure 3.6: The Bayesian network for the derivation graph snippet of Figure 3.5, with CPDs for EDB tuple nodes N(`L6`, `L7`) and U(`L6`, `L7`), conjunctive clause nodes r_1(`L6`, `L6`, `L7`) and r_2(`L7`, `L6`), and the disjunctive tuple node P(`L6`, `L7`).

3.2.3 Optimizations

The time taken for marginal inference on a Bayesian network depends on the size of the Bayesian network and on the sizes of the CPDs at each node of the network. This section discusses two optimizations that reduce the size of the Bayesian network.

Coreachability based constraint pruning. The derivation graph at fixpoint contains all derivable tuples. Not all derivable tuples in the derivation graph participate in deriving the alarm tuples. Therefore, this optimization removes unnecessary tuples and clauses by performing a backward pass over the set of clauses GC. We initialize the set of useful tuples $U := O$, where O is the set of alarm tuples, and repeatedly perform the following update until fixpoint:

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

Here, c_g is the consequent of clause g, and A_g is the set of all antecedents of g. Informally, a tuple is useful if it is either itself an alarm, or can be used to produce a useful tuple. The final pruned set of clauses is defined as follows: $GC' = \{g \in GC \mid c_g \in U\}$.

Chain compression. This optimization compresses a chain of derivation steps to a single derivation step. Consider the derivation graph shown in Figure 3.3, and observe the sequence of tuples $\mathrm{P}(\mathtt{L4}, \mathtt{L2}) \rightarrow \mathrm{P}(\mathtt{L4}, \mathtt{L3}) \rightarrow \mathrm{P}(\mathtt{L4}, \mathtt{L4}) \rightarrow \mathrm{P}(\mathtt{L4}, \mathtt{L5})$. Both intermediate tuples $\mathrm{P}(\mathtt{L4}, \mathtt{L3})$ and $\mathrm{P}(\mathtt{L4}, \mathtt{L4})$ are produced by exactly one grounded constraint, and are consumed as an antecedent by exactly one clause. Furthermore, we will never need the probabilities of these tuples for ranking as neither of them is an alarm node. We may therefore rewrite the derivation graph to directly conclude that $\mathrm{P}(\mathtt{L4}, \mathtt{L2}) \rightarrow \mathrm{P}(\mathtt{L4}, \mathtt{L5})$. We formally present this optimization in Algorithm 1.

3.3 Marginal Inference

There are several methods to perform marginal inference in Bayesian networks. Techniques for exact inference are variable elimination, the junction tree algorithm [37], and symbolic techniques [28]. All these algorithms have a complexity that is exponential in the Bayesian network's tree width [49]. Techniques for approximate inference are loopy belief propagation [44, 59], sampling methods such as Gibbs sampling (a MCMC sampling algorithm), or variational methods. We choose loopy belief propagation because we need a deterministic inference technique that scales to large networks.

Belief propagation is a message-passing algorithm for performing inference on graphical models like Bayesian networks and Markov random fields. It is an exact inference algorithm on probabilistic models that have a tree structure. However, it was found that the belief propagation could be used as an approximate inference technique on general graphical models. When used as an approximate inference

Algorithm 1 $\textsc{Compress}(GC, C, O)$, where GC is the set of grounded constraints, C is the set of derived tuples, and O is the set of alarms produced by the analysis. It returns the modified set of clauses GC.

1. For each tuple t, define:

$$\begin{aligned} \textsc{Srcs}(t) &\coloneqq \{g \in GC \mid t = c_g\}, \\ \textsc{Sinks}(t) &\coloneqq \{g \in GC \mid t \in A_g\}. \end{aligned}$$

2. Construct the following set:

$$E \coloneqq \{t \in C \setminus O \mid |\textsc{Srcs}(t)| = 1 \wedge |\textsc{Sinks}(t)| = 1\}.$$

3. While E is not empty:

   (a) Pick an arbitrary tuple $t \in E$, and let $\textsc{Srcs}(t) = \{g_1\}$, and $\textsc{Sinks}(t) = \{g_2\}$.

   (b) Since $t = c_{g_1}$ and t is an antecedent of g_2, let

$$\begin{aligned} g_1 &= a_1 \wedge a_2 \wedge \cdots \wedge a_k \implies t, \text{ and} \\ g_2 &= t \wedge b_1 \wedge b_2 \wedge \ldots b_p \implies t'. \end{aligned}$$

   (c) Define a new clause, $g' = a_1 \wedge a_2 \wedge \cdots \wedge a_k \wedge b_1 \wedge b_2 \wedge \cdots \wedge b_p \implies t'$. Update $GC \coloneqq GC \cup \{g'\} \setminus \{g_1, g_2\}$, $E \coloneqq E \setminus \{t\}$, and recompute $\textsc{Srcs}$ and $\textsc{Sinks}$.

   (d) If g_1 was associated with rule r_1 with probability p_1, and g_2 was associated with rule r_2 with probability p_2, then associate g' with a new rule r' with probability $p_1 p_2$.

4. Return GC.

technique, it is called loopy belief propagation. Empirical studies show that it often converges to values that are very close to the true marginal probabilities [61]. In each round of the inference algorithm, each random variable v passes a message to all its neighbors u (both parents and children), indicating the updated belief in u given the current belief in v. Each node then computes a new belief by combining the messages received from all its neighbors. The algorithm terminates when the beliefs converge to within a factor of ϵ across two subsequent iterations. In our experiments, we use the loopy belief propagation algorithm implemented by LibDAI, an off-the-shelf inference engine [59]. In situations where loopy belief propagation sometimes does not converge, we halt the loop by setting a limit on the number of iterations that it can execute. In such cases, we return the average belief over the last 100 iterations to suppress oscillatory behavior.

3.4 Conditioning on Evidence

In this section, we illustrate how alarm probability and hence alarm ranking changes when the probabilities are conditioned on evidence. Consider the alarm probabilities and ranking in Table 3.1a. This table is ranking the alarms reported for the example code in Figure 3.1. Among the five reported alarms, alarm race(`L0`, `L7`) is a true alarm and the rest are false alarms. But the true alarm is ranked the last. Suppose we acquire evidence that the top-ranked alarm race(`L4`, `L5`) is a false alarm. We can now recompute the alarm probabilities of all the alarms conditioned on this fact. This results in the updated list of alarms shown in Table 3.1b. Observe that the belief in the closely related alarm race(`L6`, `L7`) drops from 0.324 to 0.030. Similarly, the belief in the other two closely related alarms, race(`L5`, `L5`) and race(`L7`, `L7`), also drop. Whereas, the belief in the unrelated alarm race(`L0`, `L7`) remains unchanged at 0.279. As a result, the entire family of false alarms drops in the ranking, so that the only true alarm is now at the top.

In general, if e is the observed evidence that takes the form of an event like "tuple $t' = 1$" or "tuple $t' = 0$", then, recomputing alarm probabilities entails replacing the prior belief $\Pr(t)$ for each tuple t, with the posterior belief, $\Pr(t \mid e)$. It is possible that the observed evidence could be a more complex event, for example, a conjunction of more than one observed tuple bound to its observed value. LibDAI, the library that we use to perform marginal inference allows us *observe* the values of nodes prior to performing loopy belief propagation, thus conditioning the computed probabilities on the observed evidence.

Table 3.1: Alarm ranking by probabilities, **(a)** before, and **(b)** after the evidence $\neg\,\mathrm{race}(\texttt{L4}, \texttt{L5})$. The real datarace $\mathrm{race}(\texttt{L0}, \texttt{L7})$ rises in the ranking after conditioning on evidence.

(a) $\Pr(a)$.

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

(b) $\Pr(a \mid \neg\,\mathrm{race}(\texttt{L4}, \texttt{L5}))$.

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

3.5 Configurations of the Bayesian Network

The Bayesian network that we extract from the derivation graph is a flexible probabilistic model that encodes relationships between analysis facts. When we implement an alarm ranking system with an underlying Bayesian network, there are several design choices that we can make to customize the Bayesian network to be empirically effective.

Choosing a cycle elimination algorithm. One configurable parameter is the choice of a cycle elimination algorithm that eliminates cycles in a derivation graph before it is converted to a Bayesian network. Chapter 5 discusses three approaches for cycle elimination. The approaches differ in how aggressively they discard clauses from the derivation graph, in order to eliminate cycles. While all tuples still remain derivable in all the approaches, we may lose correlations between alarms to different extents. This is because we may lose one or more derivation trees for a tuple when a clause is discarded. Therefore cycle elimination reduces the size of the derivation graph, but increases incompleteness and may destroy some alarm correlations. The size of the Bayesian network, which is directly proportional to that of the derivation graph, greatly impacts the time taken for the convergence of approximate marginal inference, and also how often there is convergence versus timeout. Therefore, we need to employ a cycle elimination algorithm that discards the least number of clauses while ensuring the scalability of inference over the resulting Bayesian network.

Choosing optimizations. Another configurable parameter is whether or not to apply chain compression to derivation graphs. Chain compression is an optimization (discussed in Section 3.2.3) to reduce the size of the derivation graph from which a Bayesian network is constructed. Chain compression does not alter alarm probabilities only when exact inference is performed on the resulting Bayesian network. That is, exact marginal inference on Bayesian networks constructed from the compressed and uncompressed derivation graphs will produce the same probability distribution for alarms. But approximate marginal inference will not. In our setting, Bayesian networks are large and exact inference is infeasible. While chain compression helps to reduce the size of the Bayesian network, approximate inference on optimized and unoptimized Bayesian networks converge to different probability distributions for alarms. On an orthogonal note, chain compression may or may not effectively reduce the size of the derivation graph, depending on the form of the instantiated clauses. Chain compression is highly effective when there are long chains of derivation sequences involving clauses with single antecedents. Therefore, the decision of whether or not to apply chain compression is a configurable parameter.

Choosing causes of analysis incompleteness. As discussed in Section 3.2.2, we can define the CPD at each node of the Bayesian network. While doing so, we can treat each node as either complete by assigning a probability of 1.0, or as incomplete by assigning a probability strictly less than 1.0. In our implementation of the system

Bingo (Section 4.1), we have chosen to treat input tuples as complete and the deduction rules as incomplete. This is because of our belief that the false alarms stem *primarily* from the incompleteness of deduction rules, and not the input tuples, of the client static analyses of Bingo. The reason for this belief is that the one of the client static analyses of Bingo (datarace analysis) is path-insensitive, and the other (taint analysis) is flow-insensitive. In the implementation of the system Presto (Section 4.3), we treat (*a*) input tuples of the client analysis that are produced by underlying static analyses as incomplete, (*b*) input tuples that are extracted from program text as complete, and (*c*) deduction rules as complete. In the client static analysis of Presto, incomplete input tuples are the primary sources of false alarms and not the deduction rules. Presto's client analysis is an exception flow analysis that reports exception objects escaping via the main method, as alarms. One of the input relations it uses to compute exception flows is the call-graph edge. The call-graph edge is an output relation generated by a flow- and context-insensitive interprocedural pointer analysis that is executed before the exception flow analysis. If a call-graph edge is unrealizable, entire exception flows that involve this call-graph edge are unrealizable. Thus, an incomplete input tuple (a call-graph edge) causes the analysis to deduce an unrealizable exception flow and report a false alarm.

Chapter 4

The Alarm Ranking System

The alarm ranking system is an end-to-end system that performs the following tasks in order: (*a*) executes the static analysis on a program, (*b*) extracts the derivation graph at fixpoint, (*c*) constructs a Bayesian network from the derivation graph, (*d*) performs marginal inference on the Bayesian network, (*e*) produces a ranked list of alarms, and (*f*) (optionally) improves the alarm ranking by conditioning it on evidence. We have implemented two instances of the alarm ranking system: Bingo and Presto. We describe Bingo in Section 4.1, and its evaluation in Section 4.2. Next, we describe Presto in Section 4.3, and its evaluation in Section 4.4. Lastly, in Section 4.5, we analyze the ranking methodology.

4.1 End-to-End System: Bingo

The system Bingo is an alarm ranking system as described above. Its effectiveness lies in leveraging user feedback to iteratively improve the alarm ranking by conditioning the alarm probabilities on the user feedback accrued thus far. In each iteration, Bingo presents the alarm with highest probability for inspection by the user. The user then indicates its ground truth, and Bingo incorporates this feedback as evidence for subsequent iterations. We summarize this process in Figure 4.1.

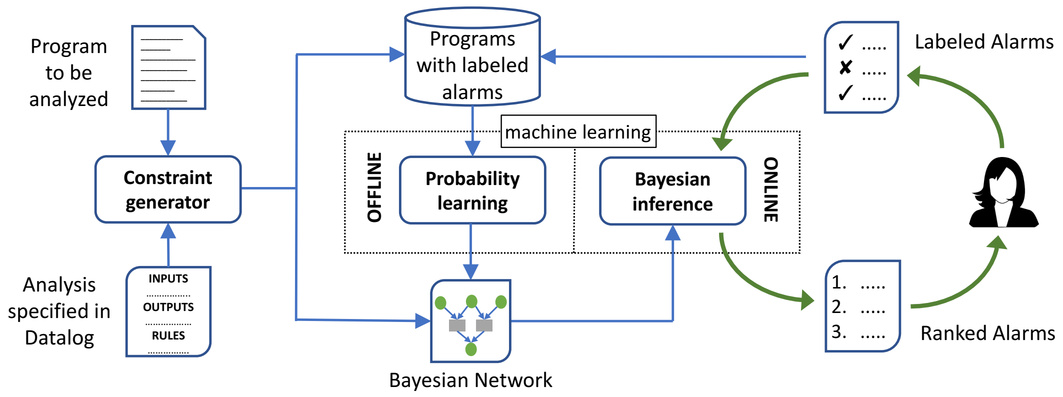


Figure 4.1: The Bingo workflow and interaction model.

In Bingo, we parameterize the Bayesian network with a vector of rule probabilities $\boldsymbol{p}$. We uniformly initialize each rule with a probability of 0.999. Ideally, we can learn these initial probabilities from labelled data. Indeed, the Bingo workflow shows a probability learning module that incorporates the learning of rule probabilities in the workflow. If we had fully labelled data (i.e., if we knew whether the consequent tuple produced by every instance of every rule, was complete or not), the rule probability is simply the fraction of rule instances producing the complete consequent. However, typical labelled data only comprises labelled alarms. That is, we will usually not have labels for all consequent tuples produces by all rule instances: they are latent or unobserved. In such situations, we could learn the rule probabilities as described in Section 4.1.1. However, this technique requires a large corpus, and a proper experimental evaluation involves partitioning the data into training and test sets. To avoid this problem, Bingo opts to parameterize the Bayesian network by assigning a uniform constant probability to each rule.

We formally describe the workflow of Bingo in Algorithm 2. For cycle elimination in step 3, Bingo uses the aggressive algorithm in Section 5.2 because it needs to scale to large programs ($\approx$ 1–10 million grounded constraints). Step 4 performs both the coreachability and the chain compression optimizations discussed in Section 3.2.3.

Algorithm 2 BINGO$(D, P, \boldsymbol{p})$, where D is the analysis expressed in Datalog, P is the program to be analyzed, and $\boldsymbol{p}$ maps each analysis rule r to its firing probability p_r.

```
1. Let $I = \textsc{InputRelations}_D(P)$. Populate all input relations $I$ using the program
   text and prior analysis results.
2. Let $(C, O, GC) = \textsc{DatalogSolve}(D, I)$. $C$ is the set of derived tuples, $O \subseteq C$ is
   the set of alarms produced, and $GC$ is the set of grounded constraints.
3. Compute $GC_c \coloneqq \textsc{CycleElimAggressive}(I, C, GC)$. Eliminate cycles from the
   grounded constraints.
4. (Optionally,) Update $GC_c \coloneqq \textsc{Optimize}(I, GC_c, O)$. Reduce the size of the set of
   grounded constraints.
5. Construct Bayesian network $BN$ from $GC_c$ and $\boldsymbol{p}$, and let Pr be its joint probability
   distribution.
6. Initialize the feedback set $\boldsymbol{e} \coloneqq \emptyset$.
7. While there exists an unlabelled alarm:
    (a) Let $O_u = O \setminus \boldsymbol{e}$ be the set of all unlabelled alarms.
    (b) Determine the top-ranked unlabelled alarm:
            $o_t = \arg\max_{o \in O_u} \Pr(o \mid \boldsymbol{e}).$
    (c) Present $o_t$ for inspection. If the user labels it a true alarm, update $\boldsymbol{e} \coloneqq \boldsymbol{e} \cup \{o_t\}$.
        Otherwise, update $\boldsymbol{e} \coloneqq \boldsymbol{e} \cup \{\neg o_t\}$.
```

BINGO uses an off-the-shelf solver [59] for the conditional probability queries in step 7(b).

We have used BINGO to rank alarms reported by two static analyses: datarace and taint analysis. We briefly describe these analyses in Section 4.1.2.

4.1.1 Getting Initial Probabilities

We describe one approach to learning good initial probabilities that quantify the incompleteness of analysis rules. Given a program and its associated ground truth $\boldsymbol{v}$, the learning problem is to determine the most likely probability vector $\boldsymbol{p}$ that explains $\boldsymbol{v}$. The *likelihood*, $L(\boldsymbol{p}; \boldsymbol{v})$ is the probability of $\boldsymbol{v}$ under the probability vector $\boldsymbol{p}$. That is, $L(\boldsymbol{p}; \boldsymbol{v}) = \Pr_{\boldsymbol{p}}(\boldsymbol{v})$. We need to estimate the probability vector that will maximize this likelihood.

$$\tilde{\boldsymbol{p}} = \arg\max_{\boldsymbol{p}} L(\boldsymbol{p}; \boldsymbol{v}). \tag{4.1}$$

In our setting, this estimation may be performed by the Expectation Maximization algorithm [40].

4.1.2 Client Analyses

The end-to-end system Bingo demonstrates our approach of augmenting program reasoning with probabilistic reasoning on two bug-finding static analyses written in Datalog: datarace analysis and taint analysis.

Datarace Analysis

The datarace analysis [64] finds pairs of program points that access a heap object (read/write), and that could potentially be executed in parallel by two different threads, with at least one of the accesses being a write access. The analysis is implemented in the Chord framework [63] and is built on top of the following:

1. A flow- and context-sensitive may-happen-in-parallel analysis that tracks program points that may be executed by pairs of threads potentially running in parallel.

2. A context-sensitive thread-escape analysis that computes the set of program

points potentially reading or writing to heap objects that may not be thread-local. A heap object in a multithreaded shared-memory program is thread-local when it is reachable only from at most a single thread. Only program points that access thread-escaping (non-thread-local) heap objects may potentially make a racy access to a heap object. The thread-escape analysis is guided by a technique [82] that efficiently searches a large family of abstractions to prove a query of the form: Does a specific program point read from or write to a heap object that is not thread-local? This technique either: (a) finds the cheapest heap abstraction that proves such a query, or (b) shows that no such abstraction exists. Therefore this technique helps the thread-escape analysis to eliminate program points that may access (read/write) heap objects that are definitely thread-local.

3. A pointer analysis [56] that is 3-context-and-object sensitive but flow-insensitive. This analysis is *soundy* [53], i.e., sound except for some features of Java, including exceptions and reflection (which is resolved by a dynamic analysis [13] - unsound but complete). The datarace analysis, may-happen-in-parallel analysis, and the thread-escape analysis use either or both the call-graph and the points-to information computed by the pointer analysis.

Typically, static analyses make unsound choices [53] so that they are scalable and practically useful. For example, the Chord [64] static analysis framework does not model exceptions in Java. In addition, the datarace analysis provides three unsound switches that introduce different degrees of unsoundness, in order to limit the number of reported alarms:

1. A switch to turn on a lock-set analysis that determines the set of locks that may guard each program point. The datarace analysis uses this as "must" information and unsoundly excludes all pairs of accesses that may be guarded by the same lock, from the set of pairs of potentially racing accesses.

2. A switch to exclude dataraces between two accesses in the same abstract thread. This exclusion is sound if one abstract thread corresponds to exactly one concrete thread. But this may not always be the case. The datarace analysis models each thread starting at a given program point as one abstract thread. If a program point that starts a thread is in the body of a loop, each distinct concrete thread that may be started in an iteration of the loop will be modeled by the same abstract thread. In such a scenario, excluding dataraces between accesses in the same abstract thread is unsound.

3. A switch to exclude dataraces in which at least one of the accesses is in a constructor. It is sound to exclude dataraces on the "this" object in constructors. But, it is unsound to exclude every datarace in which one of the racing accesses may happen within the body of a constructor.

While turning on these unsound switches reduces the total number of reported alarms, and the number of false positives, it does introduce false negatives.

The datarace analysis together with all its underlying analyses, comprises 102 rules and 102 relations. BINGO executes the datarace analysis in its soundest configuration by turning off the unsound switches, in order to avoid false negatives. It instruments the rules of the datarace analysis, thread-escape analysis and the may-happen-in-parallel analysis to extract the derivation graph that captures the reasoning steps employed by these analyses while deducing a datarace bug.

Taint Analysis

The taint analysis [24] computes data flows in Android programs, whose sources are Android framework methods that read sensitive data, and whose sinks are framework methods that leak sensitive data outside the device. The analysis tracks the flow of complex sensitive data captured in heap objects, and also tracks the flow of scalar data (e.g. integer values) assigned to static/instance fields and local variables. It is implemented in the Soot framework [78]. The taint analysis is built on top of a context-insensitive pointer analysis that is similar to the one used in the datarace analysis [56] except for context-sensitivity. The call-graph and points-to information generated by the pointer analysis is used by the taint analysis.

The sources of sensitive data and the points of leak (sinks) are marked by two types of annotations called the *source* and the *sink* annotations. These annotations are placed on program variables. The taint analysis in the Android framework comes with built-in *source* and *sink* annotations that were written manually by the analysis writer. The analysis propagates these annotations through the Android program in order to determine the flow of sensitive data to points of leak. The propagation rules associate each abstract heap object, scalar static/instance field, and scalar local variable with a (a) *source* annotation if it could contain the sensitive data generated at a source represented by the annotation, and, (b) *sink* annotation if the data it contains could flow to the point of leak represented by the annotation. In addition to these annotations, there are also *transfer* annotations for propagating the flow of *source* and *sink* annotations through the methods of the Android framework, that are used in lieu of analyzing the methods of the Android framework. This enables the taint analysis to analyze the source code of only the Android application and completely avoid analyzing the methods of the Android framework.

The taint analysis propagates *source* annotations on scalar variables by performing a forward inter-procedural dataflow analysis, and on heap objects using points-to sets. It propagates the *sink* annotations by performing a backward inter-procedural method-escape analysis. While performing both these analyses, the taint analysis uses the heap points-to information (pre-computed by pointer analysis) to transitively annotate all objects nested within an already-annotated object. The taint analysis concludes that there is a tainted data flow from an annotated source to an annotated sink if there is some abstract heap object, or some local/static/instance scalar variable that is associated with both the source annotation and the sink annotation.

The taint analysis together with all its underlying analyses, comprise 62 rules and 77 relations. Bingo instruments the rules of the taint analysis to extract the derivation graph that captures the reasoning steps employed by the taint analysis while deducing a tainted data flow.

4.2 Bingo: Experimental Evaluation

In this section, we describe the evaluation of Bingo. We start by giving details about the benchmarks used for the evaluation ((Section 4.2.1) and the reference baselines against which we perform the measurements ((Section 4.2.2).

For the evaluation, we primarily measured two things: (*a*) effectiveness: how effective is Bingo in ranking alarms (Section 4.2.3), and (*b*) robustness: how robust is the ranking produced by Bingo, to incorrect responses (Section 4.2.4). In addition to the above, we investigated if Bingo helped in discovering new bugs missed by existing precise analysis tools (Section 4.2.5), and to what extent the optimizations of Bingo helped to scale its applicability to large programs (Section 4.2.6).

4.2.1 Benchmarks

We evaluated Bingo on the suite of 16 benchmarks shown in Table 4.1. Eight of these are benchmarks to which we applied the datarace analysis, and the remaining eight are benchmarks to which we applied the taint analysis. The first four datarace benchmarks are commonly used in previous work [23, 81]. The remaining four are from the DaCapo benchmark suite [10]. We obtained the ground truth by manual inspection. The eight taint analysis benchmarks were chosen from the STAMP [24] repository, and are a combination of apps provided by a major security company, and challenge problems used in past research. Table 4.2 gives various metrics that indicate the size of the benchmarks.

Table 4.1: Benchmark description.

	Program	Description
Datarace analysis	`hedc`	Web crawler from ETH
	`ftp`	Apache FTP server
	`weblech`	Website download/mirror tool
	`jspider`	Web spider engine
	`avrora`	AVR microcontroller simulator
	`luindex`	Document indexing tool
	`sunflow`	Photo-realistic image rendering system
	`xalan`	XML to HTML transforming tool
Taint analysis	`app-324`	Unendorsed Adobe Flash player (leaks phone number and SMS content)
	`nsounds`	Music player (leaks location information)
	`app-ca7`	Simulation game (leaks phone number)
	`app-kQm`	Puzzle game (leaks phone number)
	`tmazes`	Puzzle game (packages the Mobishooter malware)
	`atrail`	RPG game (contains malicious behaviors: SD-card overwrite and delete SMS)
	`gmaster`	Image processing tool (packages the gingermaster malware on Android 2.3)
	`app-018`	Arcade game (leaks IMEI and IMSI information)

Table 4.2: Benchmark characteristics. 'Total' and 'App' columns are numbers using 0-CFA call graph construction, with and without the JDK for datarace analysis benchmarks, and with and without the Android framework for taint analysis benchmarks.

	Program	# Classes		# Methods		Bytecode (KLOC)	
		Total	App	Total	App	Total	App
Datarace analysis	`hedc`	357	44	2,154	230	141	11
	`ftp`	499	119	2,754	608	152	23
	`weblech`	579	56	3,344	303	167	12
	`jspider`	362	113	1,584	426	95	13
	`avrora`	2,080	1,119	10,095	3,875	369	113
	`luindex`	1,168	169	7,494	1,030	317	47
	`sunflow`	1,857	127	12,934	967	616	53
	`xalan`	1,727	390	12,214	3,007	520	120
Taint analysis	`app-324`	1,788	81	6,518	167	40	10
	`nsounds`	1,418	119	4,323	500	52	11
	`app-ca7`	1,470	142	4,928	889	55	23
	`app-kQm`	1,332	105	4,114	517	68	31
	`tmazes`	2,462	547	7,034	2,815	77	35
	`atrail`	1,623	339	5,016	1,523	81	44
	`gmaster`	1,474	159	4,500	738	82	39
	`app-018`	1,840	275	5,397	1,389	98	50

4.2.2 Baselines

We compare Bingo to two baseline algorithms, BaseR and BaseC. In each iteration, BaseR chooses an alarm for inspection uniformly at random from the pool of unlabelled alarms. The algorithm BaseC is based on the alarm classifier Eugene [55]. Eugene classifies alarms as true or false based on available feedback on a subset of alarms. In each iteration, (a) we invoke Eugene to classify the current set of unlabelled alarms, and (b) we pick a true alarm (as classified by Eugene) at random, for inspection and labelling by a user. When BaseC exhausts all alarms classified as true by Eugene, it picks an alarm classified as false, at random, and proposes it for inspection.

A more formal description of algorithm BaseC is given below: Let A be the set of alarms produced by the analysis. Eugene generalizes feedback on some disjoint subsets of alarms, $L_p, L_n \subseteq A$, and classifies *all* $a \in A$ as likely true and likely false: A_p, A_n, such that $A_p \cup A_n = A$, $L_p \subseteq A_p$, and $L_n \subseteq A_n$. We constructed the user-guided alarm-ranking algorithm BaseC, from Eugene, as follows:

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

4.2.3 Measuring Effectiveness

We measure the effectiveness of Bingo by the following three metrics explained below:

1. The rank at which the last true alarm is discovered (Rank-100%-T).

2. The rank at which 90% of the true alarms are discovered (Rank-90%-T). We measure this because the last true alarm to be discovered may be an outlier and therefore, may not be a conclusive indicator of the effectiveness of Bingo.

3. The area under the ROC curve (AUC). The ROC curve for a benchmark captures the dynamical behavior of the interaction process. In other words, it captures how early on in the interactive process, all true alarms are discovered. The ROC curve for a benchmark, is a graph that represents the number of false alarms on the x-axis and the number of true alarms on the y-axis, as observed at any point in the interaction process. Each point (x, y) on the ROC curve indicates a step in the interaction process where the user has inspected x false alarms and y true alarms. At each step, if the next inspected alarm is true, the next point on the ROC curve is $(x, y + 1)$. If the next inspected alarm is false, the next point on the ROC curve is $(x + 1, y)$. AUC is the (normalized) area under the ROC curve. Intuitively, the farther away the curve is above the diagonal, earlier the true alarms seen in the interaction process. That is, larger the AUC, earlier the true alarms seen in the interaction process. Therefore the AUC is a succinct metric that measures the goodness of a ranking.

We present our measurements of all the three metrics for Bingo and the two baselines, in Tables 4.3 and 4.4. For instance, the datarace analysis produces 522 alarms on `ftp`, of which 75 are real dataraces. Bingo presents all true alarms for inspection within just 103 rounds of interaction, compared to 368 for BaseC, and 520 for BaseR. Another notable example is `luindex` from the DaCapo suite, on which the analysis produces 940 alarms. Of these alarms, only 2 are real dataraces, and Bingo reports both bugs within just 14 rounds of interaction, compared to 101 for BaseC and 587 for BaseR. Over all benchmarks, on average, a user needs to inspect 44.2% and 58.5% fewer alarms than BaseC and BaseR respectively.

Table 4.3: Summary of metrics for the effectiveness of Bingo. Rank-100%-T and Rank-90%-T are the ranks at which all and 90% of the true alarms have been inspected, respectively. For the baselines, we show the median measurement across five runs. TO stands for timeout.

	Program	#Alarms			Rank-100%-T			Rank-90%-T		
		Total	Bugs	%TP	Bingo	BaseC	BaseR	Bingo	BaseC	BaseR
Datarace analysis	`hedc`	152	12	7.89%	**67**	121	143	**65**	115	135
	`ftp`	522	75	14.37%	**103**	368	520	**80**	290	476
	`weblech`	30	6	20.00%	**11**	16	29	**10**	15	25
	`jspider`	257	9	3.50%	**20**	128	247	**19**	101	201
	`avrora`	978	29	2.97%	**410**	971	960	**365**	798	835
	`luindex`	940	2	0.21%	**14**	101	587	**14**	101	587
	`sunflow`	958	171	17.85%	**838**	TO	952	**483**	TO	872
	`xalan`	1,870	75	4.01%	**273**	TO	1844	**266**	TO	1,706
Taint analysis	`app-324`	110	15	13.64%	**51**	104	106	**44**	89	97
	`nsounds`	212	52	24.53%	**135**	159	207	**79**	132	190
	`app-ca7`	393	157	39.95%	**206**	277	391	**172**	212	350
	`app-kQm`	817	160	19.58%	**255**	386	815	**200**	297	717
	`tmazes`	352	150	42.61%	**221**	305	351	**155**	205	318
	`atrail`	156	7	4.49%	**14**	48	117	**13**	44	92
	`gmaster`	437	87	19.91%	**267**	303	436	**150**	214	401
	`app-018`	420	46	10.95%	**288**	311	412	**146**	186	369

Table 4.4: Summary of metrics for the effectiveness of Bingo - continued. For the baselines, we show the median measurement across five runs. TO stands for timeout.

	Program	#Alarms			Area under the curve (AUC)		
		Total	Bugs	%TP	Bingo	BaseC	BaseR
Datarace analysis	`hedc`	152	12	7.89%	**0.81**	0.76	0.50
	`ftp`	522	75	14.37%	**0.98**	0.78	0.49
	`weblech`	30	6	20.00%	**0.84**	0.78	0.48
	`jspider`	257	9	3.50%	**0.97**	0.81	0.59
	`avrora`	978	29	2.97%	**0.75**	0.70	0.51
	`luindex`	940	2	0.21%	**0.99**	0.89	0.61
	`sunflow`	958	171	17.85%	**0.79**	TO	0.50
	`xalan`	1,870	75	4.01%	**0.91**	TO	0.50
Taint analysis	`app-324`	110	15	13.64%	**0.83**	0.58	0.50
	`nsounds`	212	52	24.53%	**0.89**	0.69	0.50
	`app-ca7`	393	157	39.95%	**0.96**	0.81	0.51
	`app-kQm`	817	160	19.58%	**0.93**	0.86	0.51
	`tmazes`	352	150	42.61%	**0.95**	0.79	0.50
	`atrail`	156	7	4.49%	**0.98**	0.81	0.60
	`gmaster`	437	87	19.91%	**0.84**	0.77	0.47
	`app-018`	420	46	10.95%	**0.85**	0.77	0.51

The case of `sunflow` illustrates the need for the metric Rank-90%-T. For `sunflow`, a user needs to inspect 838 of the 958 alarms produced to discover all bugs. However, the user discovers 90% of the true alarms within just 483 iterations. The more detailed comparison between Bingo and BaseC presented in Figures 4.2 and 4.3, demonstrates that Bingo has a consistently higher yield of true alarms than BaseC.

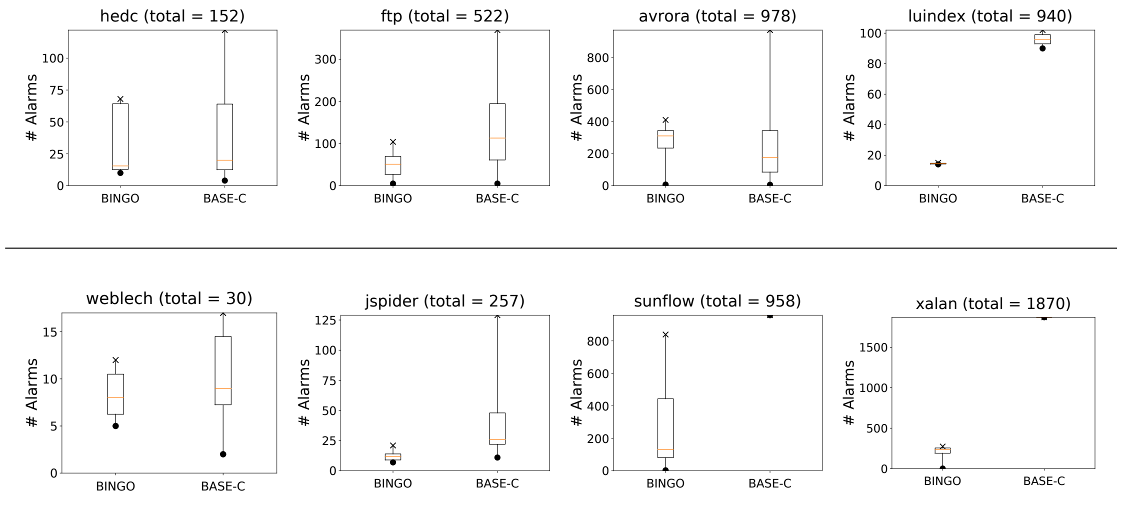


Figure 4.2: Comparing the interaction runs produced by Bingo and BaseC for the datarace benchmarks. The y-axis shows the number of alarms inspected by the user, the "●" and "×" indicate the rounds in which the first and last true alarms were discovered, and the boxes indicate the rounds in which 25%, 50%, and 75% of the true alarms were discovered. Each measurement for BaseC is itself the median of 5 independent runs.

While we provided the succinct AUC metric for all the benchmarks in the earlier tables, we show a representative and a detailed ROC plot for the `ftp` benchmark in Figure 4.4. The solid line is the ROC curve for Bingo, while the dotted lines are the ranking runs for each of the runs of BaseC, and the diagonal line is the expected behavior of BaseR. Observe that Bingo outperforms BaseC not just in the aggregate, but across each of the individual runs.

On a scale ranging from 0 to 1, on average, the AUC for Bingo exceeds that of BaseC by 0.13 and of BaseR by 0.37. In summary, we conclude that Bingo is

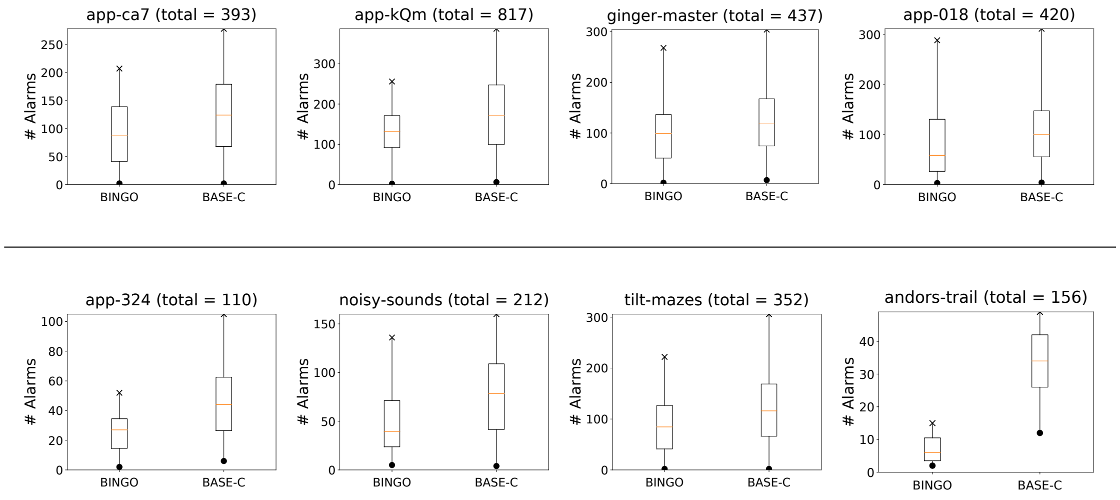


Figure 4.3: Comparing the interaction runs produced by Bingo and BaseC for the taint benchmarks. The y-axis shows the number of alarms inspected by the user, the "●" and "×" indicate the rounds in which the first and last true alarms were discovered, and the boxes indicate the rounds in which 25%, 50%, and 75% of the true alarms were discovered. Each measurement for BaseC is itself the median of 5 independent runs.

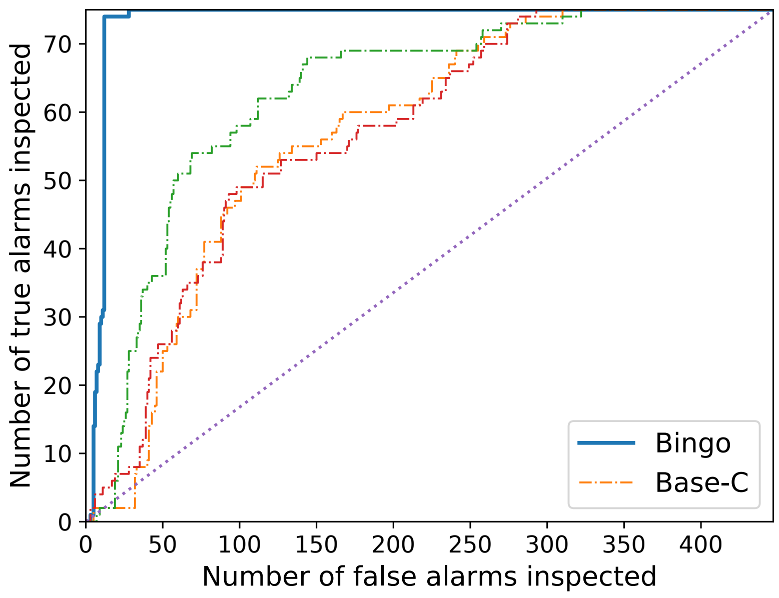


Figure 4.4: The ROC curves for `ftp`. The solid line is the curve for Bingo, while the dotted lines are the curves for each of the runs of the BaseC.

indeed effective at ranking alarms, and can significantly reduce the number of false alarms that a user needs to triage.

4.2.4 Measuring Robustness

To evaluate the robustness of Bingo, we measure the deterioration in the ranking of Bingo when it is provided with small amounts of erroneous feedback. Such a deterioration in Bingo's performance is possible because the alarm ranking it produces depends on the labelling of alarms by a human user: incorrect labelling will adversely affect ranking. First we performed a user study to estimate the approximate number of alarms that typical programmers incorrectly label, as a percentage of the total number of alarms they label.

We placed an advertisement on upwork.com, an online portal for freelance programmers. We presented respondents with a tutorial on dataraces, and gave them a 5-question test based on a small test program. Based on their performance in the test, we chose 21 of the 27 respondents, and assigned each of these developers to one of the benchmarks, `hedc`, `ftp`, `weblech`, and `jspider`. We gave them 20 alarms for labelling with an 8–10 hour time limit, such that each alarm was inspected by at least 5 independent programmers. To encourage thoughtful answers, we also asked them to provide simple explanations with their responses. We found that, for 90% of the questions, the majority vote among the responses resulted in the correct label. Equivalently, when a group of professional programmers are made to vote on the ground truth of an alarm, they incorrectly label 10% of the alarms.

We extrapolated the results of this study and simulated the runs of Bingo on `ftp` where the feedback labels had been corrupted with noise. In Table 4.5, we measure the ranks at which 90% and 100% of the alarms labelled true appear, when we corrupted feedback labels with 1%, 5% and 10% noise respectively. As is expected of an outlier, the rank of the last true alarm degrades from 103 in the original setting to 203 in the presence of noise, but the rank at which 90% of the true alarms have been inspected increases more gracefully, from 80 originally to 98 in the presence of 10% noise. In all cases, BINGO outperforms the original BASEC. We conclude that BINGO can robustly tolerate reasonable amounts of user error.

Table 4.5: Robustness of BINGO with varying amounts of user error in labelling alarms for the `ftp` benchmark. Each value is the median of three measurements.

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

4.2.5 Discovering New Bugs

Here we primarily investigate if BINGO helps to discover *new* bugs that are actually bugs missed by precise static and dynamic analysis tools (i.e., the false negatives of these tools). This would establish BINGO as a viable alternative to programmers who prefer to use precise program analysis tools that promise low false positive rates [30], despite the fact that such precise tools miss detecting many true bugs. To investigate this, we ran two state-of-the-art precise datarace detectors: (*a*) a static datarace detector in the Chord [64] framework with unsound flags turned on to increase precision, and (*b*) FastTrack [26], a dynamic datarace detector based on the happens-before relation. We ran FastTrack with the inputs that were supplied with the benchmarks.

We present the number of alarms produced and the number of bugs missed by each analyzer in Table 4.6. For example, by turning on the unsound options, we reduce the number of alarms produced by Chord from 522 to 211, but end up missing 39 real dataraces. Using BINGO, however, a user discovers all true alarms within just

Table 4.6: The number of real dataraces missed by Chord's datarace analysis with unsound settings, and the FastTrack dynamic datarace detector, for 8 Java benchmarks. New bugs are the real dataraces proposed by BINGO but missed by both. LTR is the rank at which BINGO discovers all true alarms.

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

103 iterations, thereby discovering 108% more dataraces while inspecting 51% fewer alarms.

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

Furthermore, the analysis flags determine the number of alarms produced in an unpredictable way: reducing it from 958 alarms to 506 alarms for `sunflow`, but from 1,870 alarms to 80 alarms for `xalan`. In contrast, BINGO provides the user with much more control over how much effort they would like to spend to find bugs.

4.2.6 Impact of Optimizations on Scalability

The optimizations performed on the derivation graph as explained in section 3.2.3 reduce its size greatly. This makes BINGO scale to large programs. In this section we perform measurements to quantify this reduction. We present measurements of the running time of one iteration of BINGO and of BASEC in Tables 4.7 and 4.8, with

Table 4.7: Sizes of the Bayesian networks processed by Bingo, and of the MaxSAT problems processed by BaseC, and their effect on iteration time, for the datarace benchmarks. The column heading #Tup is the number of tuples, #Cl is the number of clauses, and #Var is the number of variables, all measured in kilos (K). The heading IterT is the iteration time in seconds.

Program	Bingo, optimized			Bingo, unoptimized			BaseC		
	#Tup	#Cl	IterT	#Tup	#Cl	IterT	#Vars	#Cl	IterT
`hedc`	2.2	3.1	46	753	789	12,689	1,298	1,468	194
`ftp`	25	40	1,341	2,067	2,182	37,447	2,859	3,470	559
`weblech`	0.31	0.38	3	497	524	7,950	1,498	1,718	290
`jspider`	9	15	570	1,126	1,188	12,982	1,507	1,858	240
`avrora`	11	22	649	1,552	1,824	47,552	2,305	3,007	1,094
`luindex`	5.3	6.5	41	488	522	9,334	1,584	1,834	379
`sunflow`	59	96	3,636	9,632	11,098	timeout	26,025	34,218	timeout
`xalan`	19	32	489	2,452	2,917	51,812	6,418	8,660	timeout

Table 4.8: Sizes of the Bayesian networks processed by Bingo, and of the MaxSAT problems processed by BaseC, and their effect on iteration time, for the taint benchmarks. The column heading #Tup is the number of tuples, #Cl is the number of clauses, and #Var is the number of variables, all measured in kilos (K). The heading IterT is the iteration time in seconds.

Program	Bingo, optimized			Bingo, unoptimized			BaseC		
	#Tup	#Cl	IterT	#Tup	#Cl	IterT	#Vars	#Cl	IterT
`app-324`	0.39	1.6	83	29	1,033	14,710	129	1,178	86
`nsounds`	0.73	1.9	41	36	277	1,204	78	407	39
`app-ca7`	1.6	4.9	72	90	1,367	1,966	161	1,528	123
`app-kQm`	2.9	10	234	186	3,978	7,742	316	4,495	311
`tmazes`	1.3	3.6	57	69	1,047	2,843	153	1,198	84
`atrail`	0.42	1.3	1	13	72	183	74	145	17
`gmaster`	1.6	7.8	418	158	3,274	7,335	315	3,857	303
`app-018`	2.3	9.9	302	223	4,950	20,622	486	6,803	426

and without optimizations. The iteration time corresponds to one run of the belief propagation algorithm, and is directly dependent on the size of the Bayesian network. We indicate this size by the columns labelled #Tup and #Cl. In contrast, BaseC invokes a MaxSAT solver in each iteration, and the columns labelled #Vars and #Cl indicate the size of the formula presented to the solver. Observe the massive gains in performance—on average, an improvement of 265×—as a result of the co-reachability based pruning and chain compression, because of which Bingo can handle even large benchmark programs such as `xalan` and `sunflow`, on which BaseC times out.

4.3 End-to-End System: Presto

The system Presto is another implementation of the alarm ranking system. We instantiate Presto to rank the alarms reported by an exception flow analysis, which is described in detail in Section 4.3.1. In order to compute the probability of alarms reported by the exception flow analysis, Presto leverages the completeness of dynamic analysis.

The exception flow analysis finds the exception objects that may escape from the main method of an analyzed program, and reports them as alarms. In order to compute this, the exception flow analysis relies on the facts deduced by three other underlying analyses applied to the analyzed program. Presto applies probabilistic reasoning only to the derivations of the exception flow analysis and does not (probabilistically) analyze the other underlying analyses. Instead, Presto treats the outputs of the underlying analyses as input relations of the exception flow analysis. While constructing the probabilistic model of the exception flow analysis, Presto views the tuples of each relation input to the exception flow analysis as either complete (known with certainty), or incomplete (probabilistic). Tuples of input relations that are read off the program text are treated as complete. Tuples of input relations that have been produced by underlying analyses are treated as incomplete. This is because the underlying analyses producing these relations may be potentially incomplete. The specific input relations that are treated as incomplete is described in Section 4.3.2. Presto hypothesizes that an alarm is at least as incomplete as the analysis facts it is premised upon. In order to compute the probability of an alarm, Presto seeks probability estimates from a dynamic analysis, for the incomplete input tuples on which the alarm depends. More on how the dynamic analysis estimates these probabilities is described in Section 4.3.3. To compute alarm probabilities, Presto propagates the probabilities of input tuples by performing marginal inference on a Bayesian network extracted from the derivation graph of the exception flow analysis.

We show the workflow of Presto in Figure 4.5. The following subsections

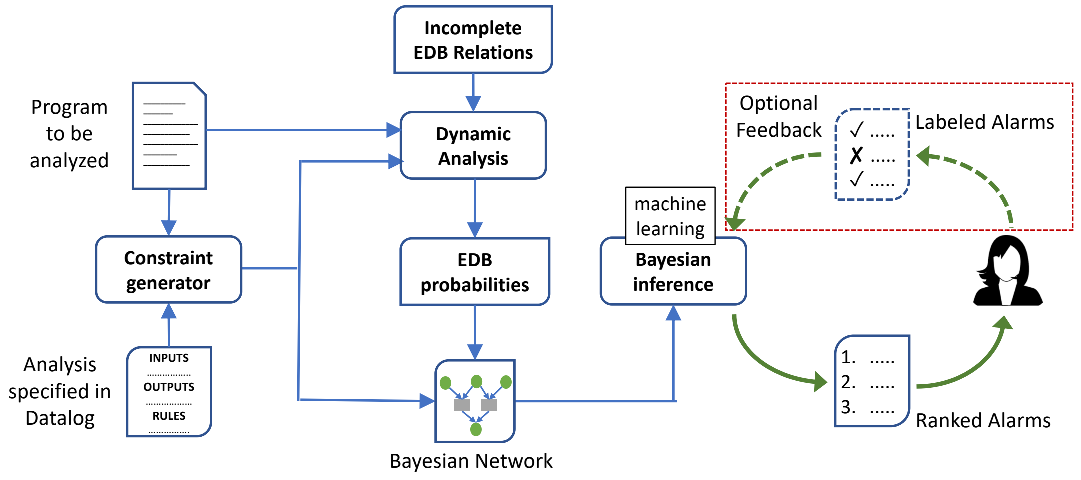


Figure 4.5: The PRESTO workflow.

elaborate each of the above aspects further. The present implementation does not recompute alarm probabilities conditioned on user feedback - it produces a one-time ranked list of alarms, and the interaction loop involving the user is not present. Moreover, seeking probability estimates for input tuples, from a dynamic analysis places its own requirements and challenges which are discussed in Section 4.3.4.

We formally describe the workflow of PRESTO in Algorithm 3. PRESTO is parameterized by a Datalog analysis D, a program P, and a set of input relation names pI that are potentially incomplete. Certain steps of the algorithm executed by PRESTO bear similarity to the steps executed by BINGO while others differ. The execution of the Datalog analysis D on program P, in steps 1 and 2 is similar in both systems.

Steps 3 and 4 are executed differently by PRESTO and BINGO. These differences are not inherent to PRESTO or BINGO. Rather, the differences are driven primarily by the characteristics of the client static analyses. The derivation graphs of the client analysis in PRESTO track deductive steps at the granularity of methods, whereas the derivation graphs of the client analyses in BINGO track deductive steps at the

Algorithm 3 PRESTO(D, P, pI), where D is the analysis expressed in Datalog, P is the program to be analyzed, and pI is the set of input relations treated probabilistically. It returns a ranked list of alarms RO.

1. Let $I = \textsc{InputRelations}_D(P)$. Populate all input relations I using the program text and prior analysis results.
2. Let $(C, O, GC) = \textsc{DatalogSolve}(D, I)$. C is the set of derived tuples, $O \subseteq C$ is the set of alarms produced, and GC is the set of grounded constraints.
3. Compute $GC_c \coloneqq \textsc{CycleElimDfs}(I, C, GC)$. Eliminate cycles from the grounded constraints.
4. (Optionally,) Update $GC_c \coloneqq \textsc{Optimize}(I, GC_c, O)$. Reduce the size of the set of grounded constraints.
5. Compute $\boldsymbol{pi} = \textsc{DynamicAnalysis}(P, GC_c, pI)$. $\boldsymbol{pi}$ is the probability vector for input tuples of the relations in set pI, that occur in some clause of GC_c.
6. Construct Bayesian network BN from GC_c and $\boldsymbol{pi}$, and let Pr be its joint probability distribution.
7. $RO = \textsc{Sort}(O, \Pr)$. Produce a ranked list of alarms RO by sorting on $\Pr(o)$ for $o \in O$, by decreasing probability.
8. Return RO.

granularity of program points. This makes the derivation graphs processed by BINGO extremely large ($\approx$ 1–10 million grounded constraints) in comparison. Therefore, BINGO uses aggressive cycle elimination to help reduce the size of the derivation graph in addition to eliminating cycles. Whereas, it is feasible for PRESTO to use the more precise dfs-based algorithm of Section 5.3 for cycle elimination in step 3. Step 4 in PRESTO performs only the coreachability optimization, and not chain compression (Section 3.2.3). BINGO performs both the optimizations. This is because derivation graphs processed by BINGO have chains of deductive steps that are amenable to effective chain compression, unlike the derivation graphs processed by PRESTO.

In step 5 of Algorithm 3, PRESTO seeks probabilities for incomplete input tuples that participate in the derivation of alarms, from a dynamic analysis. Section 4.3.3

describes how the dynamic analysis computes these probabilities. Step 6 uses an off-the-shelf solver [59] to compute the joint probability distribution.

4.3.1 Client Analysis

Exception Flow Analysis

The client analysis for Presto is a static exception flow analysis that is performed on .NET executables containing MSIL (Microsoft Intermediate Language) bytecode. Programs raise exceptions (i.e., create and throw exception objects) to flag error conditions encountered during execution. These exception objects propagate through the program and are programmatically handled in different ways. For example, they may be: (a) stored in a field of an object for later retrieval, (b) encapsulated in other exception objects, (c) caught in a catch block, (d) caught and rethrown, or (e) propagated up the method call chain. The exception flow analysis tracks the flow of exception objects through all these operations, and reports the exception objects escaping from the executable, as alarms.

The exception flow analysis is built on top of:

1. A path-sensitive intraprocedural exception analysis that computes the set of program points in a method that can throw an exception that may escape the method (i.e., may not be caught within the method).

2. A soundy context- and flow-insensitive pointer-cum-exception analysis [14] that computes the call-graph, the points-to sets, and also the set of all exception objects that may escape from an executable. It uses the intraprocedural exception analysis to compute points-to sets involving exception objects. It is implemented in the DAFFODIL [45] framework. This analysis is sound modulo the treatment of Windows library methods: (a) some methods are soundly analyzed, (b) some methods are modeled as approximations of their original counterparts, sound w.r.t. the tracked heap objects, and (c) some methods are

modeled as no-ops. The analysis also ignores reflection and calls to Windows runtime. Additionally, it treats asynchronous method calls as synchronous. This is a sound approximation because the analysis is flow-insensitive.

3. A context-insensitive exception link analysis that computes all possible "linked" pairs, where a linked pair comprises an exception catch block and a throw statement such that an exception caught by the catch block might be rethrown by the throw statement. The exception link analysis uses the points-to information produced by the pointer-cum-exception analysis.

The exception flow analysis that is based on the above underlying analyses, is a precise but unsound computation of exception propagation paths through the call-graph. It computes all feasible paths in the call-graph through which an exception object may propagate from the method where it is created, to either: (a) methods from which it does not escape, or (b) the main method from which it may escape. To find these paths, the exception flow analysis recognizes that:

1. Methods that may "catch and rethrow", "catch, wrap and throw" or "catch, unwrap and throw" an exception object, may be intermediate nodes in its propagation path. Here, by "wrap and throw" we mean the encapsulation of an exception object within another exception object that is then thrown. By "unwrap and throw" we mean the inverse operation in which an encapsulated exception object is extracted and thrown.

2. In case of asynchronous programs, methods that may "catch and store (into a field of a heap object)" an exception object may be intermediate nodes in its propagation path because the exception object may propagate further if another method retrieves it from the field and throws it.

Therefore the exception flow analysis makes the following two unsound assumptions (corresponding to the two observations made above) while finding the exception propagation paths:

1. The analysis assumes that if an exception object is rethrown, then the rethrow happens within the lexical scope of some catch block where the exception object may be caught.

2. The analysis assumes that a field store/load of an exception object happens only because of asynchrony: An exception object is stored into a field of a Task [1] object only by an asynchronous method, and it is retrieved from that field and rethrown only by the method that awaits the completion of execution of the asynchronous method.

While finding exception propagation paths in both the above scenarios, the exception flow analysis accounts for the fact that an exception object may get wrapped, or unwrapped along a propagation path.

The only impact of the above unsound assumptions is that the propagation path of an exception object may be broken up into two or more disjoint segments, which the exception flow analysis is unable to link together. We elaborate with an example. Suppose an exception object is thrown at a program point labelled A. It propagates up the program call graph and is caught at some catch block labelled B. This catch block has a method to process exceptions, so the exception object is passed to this method. This method now throws the exception object at a program point labelled C. The exception object propagates further and escapes through the Main method. The complete propagation path for the exception object is therefore: $A \rightarrow \ ... \rightarrow B \ \rightarrow C \ \rightarrow \ ... \ \rightarrow Main$. Note that as the program point C is not within the lexical scope of the catch block B, the exception flow analysis loses the link $B \ \rightarrow C$. This results in two disjoint propagation paths for the exception object: $A \rightarrow \ ... \rightarrow B$ and $C \ \rightarrow \ ... \ \rightarrow Main$.

The exception flow analysis together with all its underlying analyses, comprises 164 rules and 151 relations. In order to rank the alarms reported by the exception flow analysis, Presto needs to associate with each exception object a probability with which it may escape from the executable. We hypothesize that this is the probability that there exists a likely feasible path in the call-graph through which the exception object may propagate from the method where it is created, to the main method from which it may escape. In order to compute this probability, Presto instruments the rules of the exception flow analysis, and treats the outputs of the analyses underlying it as incomplete input relations.

4.3.2 Incomplete EDB Tuples

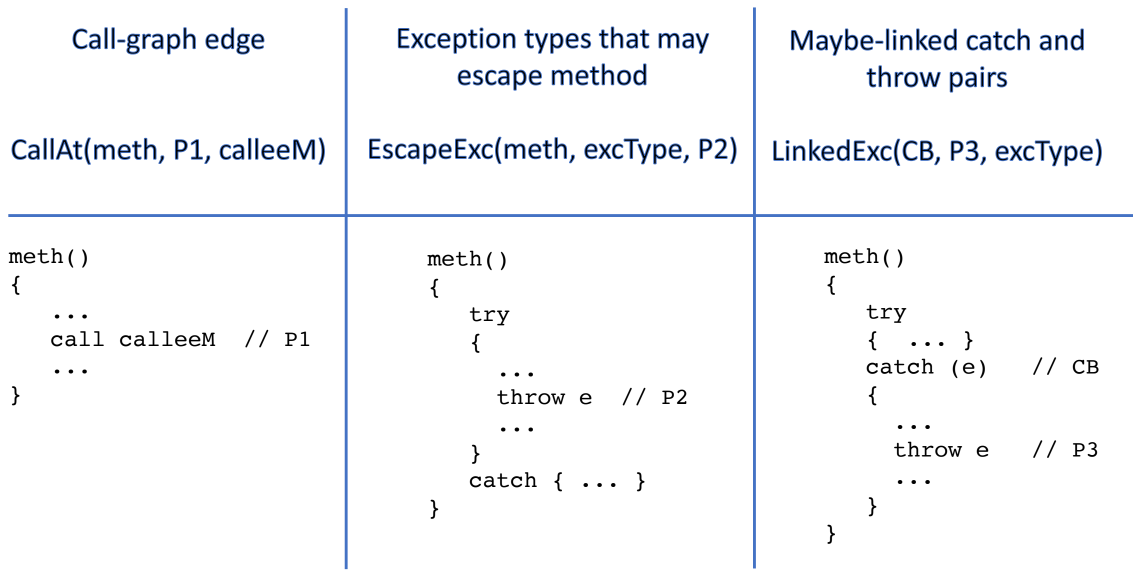
While constructing the Bayesian network from a derivation graph of the exception flow analysis, Presto treats tuples of EDB relations extracted from program text as complete (i.e., known to be true with certainty). Whereas, Presto treats the tuples of intermediate relations that are inputs to the exception flow analysis, but are outputs of the analyses underlying it, as incomplete and seeks probabilities for such tuples from a dynamic analysis.

These relations, pictorially illustrated in Figure 4.6, are:

1. The relation *CallAt* that is produced by the interprocedural pointer-cum-exception analysis. It contains tuples (*meth*, $P1$, *calleeM*) indicating that method *calleeM* may be called by the method *meth* at the program point $P1$.

2. The relation *EscapeExc* that is produced by the intraprocedural exception analysis. It contains tuples (*meth*, *excType*, $P2$) indicating that an exception of type *excType*, thrown at the program point $P2$, may escape from method *meth*. Note that the analysis producing this relation is intraprocedural, and hence it follows that the method *meth* contains the program point $P2$.

3. The relation *LinkedExc* that is produced by the exception link analysis. It contains tuples (CB, $P3$, *excType*) indicating that a catch block CB may catch an exception of type *excType* that may be rethrown at the program point $P3$. Note that while it is possible for CB and $P3$ to belong to different methods,


Call-graph edge
CallAt(meth, P1, calleeM)
Exception types that may escape method
EscapeExc(meth, excType, P2)
Maybe-linked catch and throw pairs
LinkedExc(CB, P3, excType)

```
meth()
{
   ...
   call calleeM   // P1
   ...
}
```

```
meth()
{
   try
   {
     ...
     throw e   // P2
     ...
   }
   catch { ... }
}
```

```
meth()
{
   try
   {  ... }
   catch (e)    // CB
   {
     ...
     throw e    // P3
     ...
   }
}
```

Figure 4.6: The tuples of the EDB relations that PRESTO treats as incomplete. The relation *CallAt* says that method *meth* may call method *calleeM* at program point $P1$, the relation *EscapeExc* says that the statement at program point $P2$ may throw an exception of type *excType* that may escape from method *meth*, and finally the relation *LinkedExc* says that an exception of type *excType* that may be thrown by the statement at program point $P3$ is related to the exception that may be caught by the catch block CB.

such a programming style is uncommon among programmers and therefore PRESTO does not handle this case, as explained in Section 4.3.1.

In addition to the above relations, there are three other important relations that are produced by the pointer-cum-exception analysis, that are used in the exception link analysis and also the exception flow analysis. These are the points-to relations: variable points-to, static field points-to and instance field points-to. PRESTO conservatively treats the tuples of these relations as complete even though they are, arguably, incomplete. This is because these relations involve abstract heap objects each of which may correspond to multiple concrete heap objects. Therefore trying to estimate probabilities for such tuples will make the dynamic analysis extremely complex, and practically infeasible.

4.3.3 EDB Tuple Probabilities from Dynamic Analysis

Inspired by the approach taken in previous work [71, 7], we proceed to define a way of estimating the probability of an input tuple. We associate each input relation with two "states" of program execution that are observable by a dynamic analysis. We shall refer to these two states as the "pre-state" and the "post-state". For example, for the input tuple $\mathit{CallAt}(\mathit{meth},\ P1,\ \mathit{calleeM})$:

1. the pre-state is "program execution reaches $P1$ in method *meth*", and

2. the post-state is "method *calleeM* is called at $P1$".

Note that the pre-state and the post-state are associated with an input relation, but are parameterized by the arguments of the tuples in that relation. We interpret $\Pr(t)$ for an input tuple t, as the probability that the program execution reaches the post-state, *assuming* that program execution is in the pre-state. This interpretation is justified in [71] in which it is an essential assumption that the probability of the program execution following a particular branch is independent of the execution history. However, such an assumption is necessary to get a handle on the problem being tackled.

We now specify the pre-state and post-state for the other two input relations treated probabilistically by PRESTO, namely, *EscapeExc* and *LinkedExc*.

1. For a tuple t of the form $\mathit{EscapeExc}(\mathit{meth},\ \mathit{excType},\ P2)$:

    (a) the pre-state is "an exception of type *excType* is raised at program point $P2$ of method *meth*", and

    (b) the post-state is "an exception of type *excType* escapes from method *meth*".

2. For a tuple t with form $LinkedExc(CB,\ P3,\ excType)$:

   (a) the pre-state is "the catch block CB catches an exception of type $excType$", and

   (b) the post-state is "the program point $P3$ throws an exception of type $excType$".

One immediate concern arises: it is very difficult to realize the above pre-states during program execution because it is hard to find program inputs that trigger exceptions at specific program points. The dynamic analysis that PRESTO works with, needs to have the capability to *inject* exceptions. With this capability, a dynamic analysis will be able to observe the above pre- and post-states.

For each incomplete input tuple t, the dynamic analysis counts the number of times it observes the pre-state and post-state over all the program executions that it is able to observe, and records the following:

1. $freq_{t_{pre}}$: the number of times the dynamic analysis observed the program execution reaching the pre-state, and

2. $freq_{t_{post}}$: the number of times the dynamic analysis observed the program execution reaching post-state, given that the program execution was already in the pre-state.

We then estimate the probability of a tuple t as follows:

$$\Pr(t) = P_{min} + (P_{max} - P_{min}) * \left(\frac{freq_{t_{post}}}{freq_{t_{pre}}}\right)^{1/K}$$

where P_{min} and P_{max} are the minimum and maximum probabilities that an input tuple can have, and K is a tunable parameter. We explain further below.

The parameter K has the effect of increasing the impact of $freq_{t_{post}}$. Alternatively, K has the effect of decreasing the impact of $freq_{t_{pre}}$ - $freq_{t_{post}}$. This is the number of times a post-state was not observed in spite of the execution being in a valid pre-state. We may need to tune the value of K in order to control this “impact” in certain situations as described next. It is generally difficult to trigger error conditions during test runs of the program, such that the post-states representing error handling code, are observed by the dynamic analysis. As a result, the dynamic analysis may observe a high number of instances in which the post-states were not observed in spite of the pre-states being observed. In the context of PRESTO, a post-state examines exception handling in a situation when an error may occur. Such missed observations of a post-state will decrease the probability of an exception escaping the executable. Under the assumption that an error occurs, we want this probability to be high if the exception indeed escapes the executable. Therefore, we need to increase the “impact” of the post-states that have been observed, as compared to the post-states that have not been observed. We call the parameter K as the *impact factor* because it helps to produce this effect.

The values P_{min} and P_{max} are required because a dynamic analysis is not sound - in general, it is not possible for a dynamic analysis to observe every possible path during program execution. Therefore, $P_{min} > 0$ and $P_{max} < 1$. All of P_{min}, P_{max} and K are tunable parameters that will impact the ranking of alarms and they need to be configured to be empirically effective.

4.3.4 Dynamic Analysis: Requirements and Challenges

This section discusses the requirements placed on a dynamic analysis by PRESTO, and the challenges faced by the dynamic analysis in satisfying these requirements. This discussion applies to any dynamic analysis that may interface with PRESTO. To evaluate PRESTO, we used a proprietary dynamic analysis provided by Microsoft Research, Redmond, that had the necessary capabilities to interface with PRESTO.

Requirements

For each kind of observation that a dynamic analysis has to make, instrumentation customized to the observation needs to be inserted into the program executable. For example, to track method calls, a dynamic analysis needs to insert instrumentation to log at all program points where method calls are made. A more complex example is when a dynamic analysis has to inject a fault (in the context of Presto, an exception) and observe its effects. Typical fault injection requirements placed by Presto have two forms:

1. When a program statement S in method A throws an exception of type T, a dynamic analysis needs to observe if this exception escapes from method A. Here, the program statement S is either a throw statement or a method call.

2. When a catch block CB catches an exception of type T, a dynamic analysis needs to observe if a throw statement A within the lexical scope of the catch block CB, throws this exception. Here, in order to ensure that the catch block CB catches an exception of type T, there is an implicit requirement for some statement in the try block associated with the catch block CB, to throw an exception of type T.

In order to realize such scenarios during program execution, and make the corresponding observations, a dynamic analysis needs to insert instrumentation both in the caller, and in the callee methods involved in a scenario.

Challenges

A dynamic analysis faces two primary challenges in order to satisfy the above requirements:

1. The challenge of instrumenting large libraries. The programs that Presto analyzes link with a large number of massive framework libraries. In order

to log all method calls, nearly every method except the leaf methods, in all the linked libraries needs to be instrumented. This entails huge costs in terms of instrumentation time, memory footprint of the instrumented assemblies, and a potentially unacceptable degradation in the instrumented program's performance. To resolve this issue, dynamic analyses typically find a tradeoff point that entails instrumenting only specified assemblies. The impact of this for PRESTO is that the dynamic analysis will not be able to provide probability estimates for tuples that refer to uninstrumented code.

2. The challenge of constructing an exception object to inject. In order to provide probability estimates for certain kinds of tuples, the dynamic analysis needs to insert instrumentation that throws an exception object. Such an exception object may potentially be a complex object with nested exceptions and many other reference fields. It is non-trivial to construct such an object. To resolve this issue, dynamic analyses may construct exception objects using default constructors or rely on the user to provide all the necessary information. The impact of this for PRESTO is that if the dynamic analysis is allowed to construct exception objects using default constructors, then the observations it makes may not match actual concrete program executions. If PRESTO has to provide all necessary information, then it needs more sophisticated program analyses to automatically gather the relevant information.

Apart from the challenges above that are faced by a dynamic analysis, PRESTO faces a challenge in using such dynamic analyses to get probability estimates for tuples. It is the challenge of scalability: PRESTO needs to execute the dynamic analysis several times. For tracking method calls, one execution of the dynamic analysis for each available program input, suffices. The probability estimates for all call-graph tuples can be extracted from this set of executions. Whereas, one such set of executions of the dynamic analysis is required for each tuple of the relations *EscapeExc* and *LinkedExc*, each of which represents a fault injection scenario. This is because a fault injection scenario corresponds to exactly one tuple. That is, for each single fault injection scenario, if we execute the dynamic analysis on all available inputs, we obtain a probability estimate for one tuple.

Two possible approaches to alleviate this problem may be:

1. Presto should not seek probability estimates from the dynamic analysis for all tuples of the relations *EscapeExc* and *LinkedExc*. It should selectively choose relevant or impacting tuples. But determining relevant or impacting tuples is non-trivial, and can be a direction for future research. A simpler alternative is to take a heuristic approach that limits the total number of tuples that represent fault injection scenarios, for which Presto seeks probability estimates, to some constant N. These tuples could be the N most frequently occurring tuples in the derivation graph.

2. Presto can adopt engineering approaches like executing various instances of the dynamic analysis in parallel on multiple machines or processors. Another option is to execute the dynamic analysis on only a subset of the available inputs. But choosing an appropriate subset of inputs is a problem in its own right.

The approach that the current implementation of Presto takes during evaluation is to manually limit program inputs to small subsets of the available inputs. This approach suffices for the current set of benchmarks as they are not stressing Presto beyond its scalability limits.

4.4 Presto: Experimental Evaluation

In this section, we describe the evaluation of Presto. We start by giving details about the benchmarks used for the evaluation ((Section 4.4.1). Unfortunately, we were unable to find any static analysis tools that found exception flow bugs for .NET executables. Hence we do not have any reference baseline against which we can calibrate the performance of PRESTO.

For the evaluation, we primarily did two things: (*a*) measured effectiveness: how effective is PRESTO in ranking alarms (Section 4.4.2), and (*b*) performed parameter tuning: what parameter values make PRESTO produce an empirically optimal ranking (Section 4.4.3). In addition to the above, we investigated to what extent the optimization employed by PRESTO helped to scale its applicability (Section 4.4.4).

4.4.1 Benchmarks

We evaluated PRESTO on the suite of 6 benchmarks shown in Table 4.9. These benchmarks are mid-sized C# programs got from public repositories like github.com and codeproject.com. For these benchmarks, the static analysis of many of the system classes of the .NET framework has been suppressed. This approach is commonly adopted for framework-based applications (Ex.: applications in the Android framework). In the context of PRESTO, two specific reasons are:

1. The Rapid Type Analysis (RTA) implementation does not scale when the analysis of many of the .NET framework classes is not suppressed.

2. Scalar types (like int, short) are represented in the .NET framework as struct types (like Int32, Int16). While in general, struct types are modeled as objects on the heap and tracked during pointer analysis, scalar types that are internally represented as struct types are not tracked. Therefore the analysis of such struct types can be suppressed.

Table 4.10 gives various metrics that indicate the size of the benchmarks. We obtained the ground truth for the alarms reported by the exception flow analysis applied to these benchmarks, by manual inspection.

Table 4.9: Benchmark description.

Program	Description
`AsyncJobDispatcher`	Asynchronous master-worker based job dispatcher using an FSM
`AsyncWebCrawler`	Asynchronous web crawler
`FilePkgUtil`	File packaging utility
`HtmlSanitizer`	White list rule-based HTML sanitizer
`ScrambledSquares`	Game solver with a backtracking algorithm
`AsyncWaveformGenerator`	Asynchronous generator of waveforms from MP4 audio file

Table 4.10: Benchmark characteristics. 'Total' and 'App' columns are numbers after RTA (Rapid Type Analysis) was performed for scope construction. The numbers specified under the 'Total' columns include both application code and the unsuppressed system libraries of the .NET framework. The numbers specified under the 'App' columns include only application code.

Program	# Classes		# Methods		Bytecode (KLOC)	
	Total	App	Total	App	Total	App
`AsyncJobDispatcher`	285	21	923	117	16.7	1.8
`AsyncWebCrawler`	217	19	637	59	14.3	1.6
`FilePkgUtil`	323	5	1526	19	34.4	0.5
`HtmlSanitizer`	237	29	1019	263	35.8	2.0
`ScrambledSquares`	171	8	601	73	13.2	1.3
`AsyncWaveformGenerator`	258	10	813	87	18.5	3.1

4.4.2 Measuring Effectiveness

We measure the effectiveness of Presto using the following two metrics:

1. The rank at which the last true alarm is discovered (Last true rank).

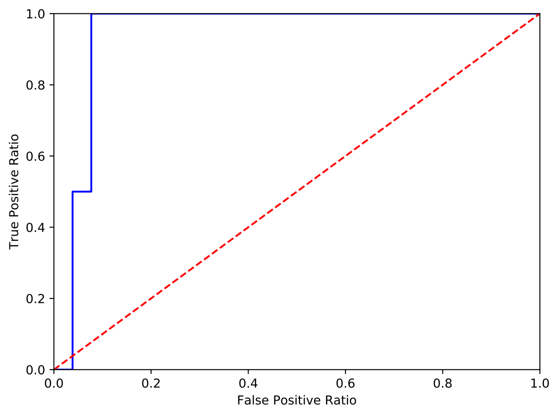
2. The area under the ROC curve (AUC). As explained in detail in Section 4.2.3, the AUC is a succinct metric that measures the goodness of a ranking. Larger the AUC, better is the ranking.

We present our measurements of both the metrics for Presto in Table 4.11. The expected value of the AUC for random ranking is 0.5. Observe that the AUC of the ranking produced by Presto for each of the benchmarks is well above 0.5. Over all benchmarks, on average, a user needs to inspect 64% fewer alarms as compared to inspecting all alarms.

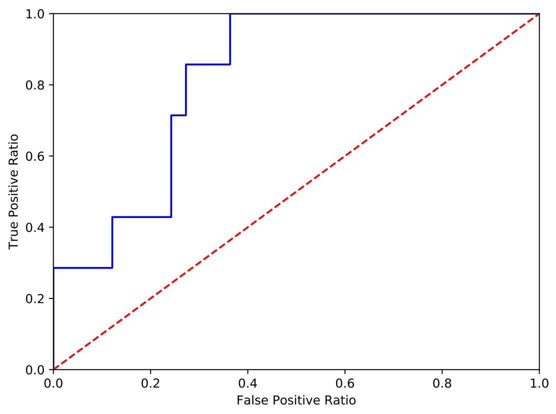
Figure 4.7 shows the ROC plots for two representative benchmarks. The dotted red line in these plots is the expected behavior of random ranking. Intuitively, the

Table 4.11: Summary of metrics for the effectiveness of PRESTO. Last true rank indicates the rank at which all of the true alarms have been inspected.

Program	Total alarms	True alarms	% True alarms	Last true rank	AUC
`AsyncJobDispatcher`	28	2	7.14%	4	0.94
`AsyncWebCrawler`	24	3	12.5%	15	0.78
`FilePkgUtil`	18	4	22.22%	7	0.89
`HtmlSanitizer`	40	7	17.5%	19	0.82
`ScrambledSquares`	15	1	6.67%	2	0.93
`AsyncWaveformGenerator`	22	4	18.18%	6	0.89



(a) `AsyncJobDispatcher`.

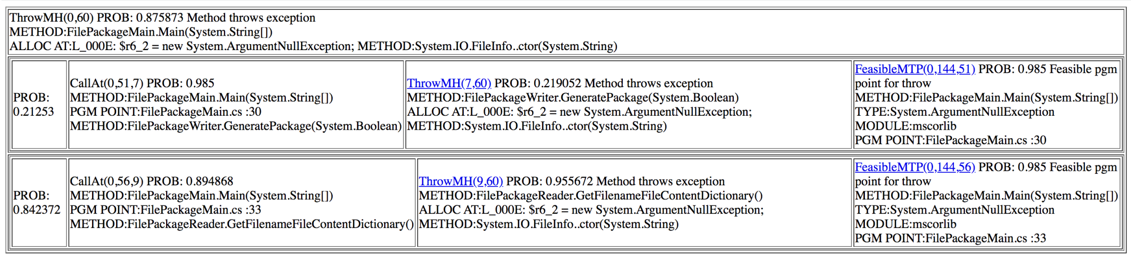


(b) `HtmlSanitizer`.

Figure 4.7: The ROC plots for two benchmarks. The diagonal dotted line in red is the expected behavior of random ranking (with expected AUC = 0.5). The AUC for `AsyncJobDispatcher` is 0.94 and the AUC for `HtmlSanitizer` is 0.82.

farther away the curve is above the diagonal, better is the ranking. Therefore, the area under this curve indicates how high up true alarms are in the ranking.

Figure 4.8 shows a snippet of the HTML reports generated by PRESTO that shows the probabilities assigned to individual tuples and derivation steps. This report snapshot shows how a tuple $ThrowMH(0, 60)$ was derived by the exception flow analysis. Tuple $ThrowMH(0, 60)$ represents the fact that an exception object that is identified by the number 60, is thrown by method $Main$ that is identified by the number 0. The report shows the two ways in which tuple $ThrowMH(0, 60)$ can be derived, in the two bottom rows. Each of these rows represents a disjunction: the body



ThrowMH(0,60) PROB: 0.875873 Method throws exception METHOD:FilePackageMain.Main(System.String[]) ALLOC AT:L_000E: $r6_2 = new System.ArgumentNullException; METHOD:System.IO.FileInfo..ctor(System.String)			

PROB:
0.21253
	CallAt(0,51,7) PROB: 0.985
	METHOD:FilePackageMain.Main(System.String[])
	PGM POINT:FilePackageMain.cs :30
	METHOD:FilePackageWriter.GeneratePackage(System.Boolean)
		ThrowMH(7,60) PROB: 0.219052 Method throws exception
		METHOD:FilePackageWriter.GeneratePackage(System.Boolean)
		ALLOC AT:L_000E: $r6_2 = new System.ArgumentNullException;
		METHOD:System.IO.FileInfo..ctor(System.String)
			FeasibleMTP(0,144,51) PROB: 0.985 Feasible pgm point for throw
			METHOD:FilePackageMain.Main(System.String[])
			TYPE:System.ArgumentNullException
			MODULE:mscorlib
			PGM POINT:FilePackageMain.cs :30

PROB:
0.842372
	CallAt(0,56,9) PROB: 0.894868
	METHOD:FilePackageMain.Main(System.String[])
	PGM POINT:FilePackageMain.cs :33
	METHOD:FilePackageReader.GetFilenameFileContentDictionary()
		ThrowMH(9,60) PROB: 0.955672 Method throws exception
		METHOD:FilePackageReader.GetFilenameFileContentDictionary()
		ALLOC AT:L_000E: $r6_2 = new System.ArgumentNullException;
		METHOD:System.IO.FileInfo..ctor(System.String)
			FeasibleMTP(0,144,56) PROB: 0.985 Feasible pgm point for throw
			METHOD:FilePackageMain.Main(System.String[])
			TYPE:System.ArgumentNullException
			MODULE:mscorlib
			PGM POINT:FilePackageMain.cs :33

Figure 4.8: Snapshot of an HTML report page generated by PRESTO that shows the probabilities assigned to individual tuples and derivation steps.

of an instantiated Datalog rule that was applied to derive tuple $ThrowMH(0, 60)$. Within each row, are the conjuncts: the tuples that comprise the body of the instantiated Datalog rule. Notice that every tuple and every disjunct is assigned a probability by PRESTO.

Delving further into this report, we observe that it indicates that the exception object identified by the number 60 may be thrown by the $Main$ method if either:

1. it is thrown by the method $GeneratePackage$ called by $Main$ at line 30, or
2. it is thrown by the method $GetFilenameFileContentDictionary$ called by $Main$ at line 33.

Observe further that PRESTO assigns a low probability of 0.21253 to the path represented by (1) above, and a much higher probability of 0.842372 to the path represented by (2) above. Studying the reports generated by PRESTO and examining the probabilities it assigns to individual facts and rules, will help a user in finding highly likely paths along which an exception might escape the program executable.

4.4.3 Tuning Parameter Values

As we saw in Section 4.3.3, the computation performed by a dynamic analysis to estimate the probability of a tuple, is parameterized by three entities:

1. the maximum probability that a tuple can have, P_{max},
2. the minimum probability that a tuple can have, P_{min}, and
3. the impact factor, K.

In this section we discuss the experiments performed to find optimal values for these parameters. We computed the AUCs for all benchmarks for all combinations of:

1. $K = \{1, 4\}$
2. $P_{max} = \{0.999,\ 0.985\}$
3. $P_{min} = \{0.5,\ 0.4,\ 0.1,\ 0.05\}$

Table 4.12 shows the results of our experiments. Based on these experiments, the optimal values of the above parameters are: $K = 4$, $P_{max} = 0.985$ and $P_{min} = 0.05$.

Table 4.12: AUC computations for all benchmarks, with different parameter values. The tunable parameters of PRESTO are: the impact factor K, the maximum and minimum probability an alarm can have, P_{max} and P_{min} respectively.

Program	Impact factor $K = 4$							
	$P_{max} = 0.999$				$P_{max} = 0.985$			
$P_{min} \rightarrow$	0.50	0.40	0.10	0.05	0.50	0.40	0.10	0.05
`AsyncJobDispatcher`	0.77	0.88	0.92	0.92	0.75	0.75	0.94	**0.94**
`AsyncWebCrawler`	0.59	0.60	0.78	0.78	0.52	0.59	0.70	**0.78**
`FilePkgUtil`	0.54	0.54	0.89	0.89	0.54	0.57	0.89	**0.89**
`HtmlSanitizer`	0.77	0.79	0.84	0.84	0.75	0.78	0.82	**0.82**
`ScrambledSquares`	0.79	0.86	0.93	0.93	0.79	0.86	0.93	**0.93**
`AsyncWaveformGenerator`	0.89	0.89	0.89	0.89	0.89	0.89	0.89	**0.89**
Program	Impact factor $K = 1$							
	$P_{max} = 0.999$				$P_{max} = 0.985$			
$P_{min} \rightarrow$	0.50	0.40	0.10	0.05	0.50	0.40	0.10	0.05
`AsyncJobDispatcher`	0.77	0.88	0.92	0.92	0.75	0.75	0.94	0.94
`AsyncWebCrawler`	0.59	0.60	0.78	0.78	0.52	0.59	0.70	0.78
`FilePkgUtil`	0.50	0.50	0.54	0.71	0.50	0.50	0.61	0.71
`HtmlSanitizer`	0.70	0.72	0.78	0.78	0.68	0.69	0.78	0.78
`ScrambledSquares`	0.79	0.86	0.93	0.93	0.79	0.86	0.93	0.93
`AsyncWaveformGenerator`	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.89

Note that on closer inspection of the measurements, we observe very interesting behaviors like:

1. Benchmarks `FilePkgUtil` and to a lesser extent, `HtmlSanitizer` are sensitive to the impact factor K. As anticipated in the discussion in Section 4.3.3, it is indeed the case that Benchmark `FilePkgUtil` has two executions to exercise error handling paths in the program, and one execution of the dynamic analysis for a typical input. In spite of this, it is necessary to increase the impact of the observations made by the dynamic analysis during the "error handling" runs of the benchmark, to get a good ranking of alarms.

2. Benchmarks `AsyncJobDispatcher`, `AsyncWebCrawler`, `FilePkgUtil`, `ScrambledSquares` and `HtmlSanitizer` are all sensitive to the value of the minimum tuple probability, P_{min}. The reason is quite interesting but concerns the nature of the exception flow analysis. The exception flow analysis has two Datalog rules whose head tuples (i.e., the consequent tuples) may potentially have a large number of disjuncts when the rules are instantiated. Consider an instantiation where all the disjuncts of a particular (head) tuple, have the minimum probability P_{min}. In spite of all disjuncts having the minimum probability P_{min}, the sheer number of disjuncts will cause the probability of the head tuple to be fairly high (intuitively, if a tuple t can be derived either by an instantiated rule r_1 or by an instantiated rule r_2, then the probability that tuple t is derivable will be greater than or equal to the larger of: (a) the probability that rule r_1 is applicable, and (b) the probability that rule r_2 is applicable). The high probability of the head tuple gets propagated along an exception flow path and may eventually result in a false alarm getting assigned a high probability, which in turn adversely affects ranking. This effect is exacerbated when the value of P_{min} itself is high (like 0.5 or 0.4). These two rules of the exception flow analysis cause the benchmarks to be particularly sensitive to the value of P_{min}.

3. Benchmarks `AsyncJobDispatcher` and `HtmlSanitizer` are slightly sensitive to

the maximum tuple probability P_{max}. The AUC of benchmark `HtmlSanitizer` worsens a little, and the AUC of benchmark `AsyncJobDispatcher` improves a little, when P_{max} changes from 0.999 to 0.985. But, on the whole, the benchmarks are not sensitive to the value of P_{max}. For a range of values of P_{max} going from 0.995 down to 0.975, the AUCs remained steady when P_{max} was decremented in steps of 0.005.

4.4.4 Impact of Optimization on Scalability

In this section, we examine the impact optimization has on PRESTO's ability to scale to larger programs. Tables 4.13 and 4.14 give the measurements.

With optimization turned on, there is a tremendous reduction in the size of the Bayesian network, and a corresponding reduction in the time required to construct the Bayesian network and perform marginal inference. Similarly, there is also great reduction in the number of runs of the dynamic analysis required, and correspondingly in the dynamic analysis time. Therefore optimizations to reduce the size of the Bayesian network are critical for the scalability of PRESTO.

However, observe that the dynamic analysis time dominates the overall end-to-end time taken by PRESTO for any benchmark. Section 4.3.4 discusses some options for alleviating this bottleneck.

Table 4.13: Scalability metrics for ranking with and without optimizations. #Clauses and #Tuples specify the size of the Bayesian network. Ranking time includes the time for constructing the Bayesian network from a derivation graph, and the time for performing marginal inference.

Program	With optimization			Without optimization		
	#Clauses (K)	#Tuples (K)	Ranking time(s)	#Clauses (K)	#Tuples (K)	Ranking time(s)
`AsyncJobDispatcher`	1.1	1.5	2.5	54.8	108.6	157
`AsyncWebCrawler`	0.7	0.9	1.5	42.6	84.6	122
`FilePkgUtil`	1.8	2.5	4.0	168.5	335.7	472
`HtmlSanitizer`	2.5	2.4	6.0	65.6	128.4	185
`ScrambledSquares`	1.0	1.3	2.0	45.5	90.2	130
`AsyncWaveformGenerator`	0.7	1.1	2.0	74.6	148.8	211

Table 4.14: Scalability metrics for dynamic analysis with and without optimizations. Dynamic analysis time includes the time for generating dynamic analysis configurations, executing the dynamic analysis, and extracting probabilities from the dynamic analysis logs. Dynamic analysis runs gives the number times the dynamic analysis was run (each time with a different configuration).

Program	With optimization		Without optimization	
	Dyn analysis runs	Dyn analysis time(m)	Dyn analysis runs	Dyn analysis time(m)
`AsyncJobDispatcher`	100	8.0	1107	95.5
`AsyncWebCrawler`	72	6.0	821	71.5
`FilePkgUtil`	135	3.5	1299	33.5
`HtmlSanitizer`	86	7.0	1937	178.5
`ScrambledSquares`	34	2.5	2725	200.0
`AsyncWaveformGenerator`	59	5.5	1142	121.5

4.5 Analysis of Alarm Ranking

Stopping criterion. Given a ranking of alarms by their probability of being true bugs, it is not clear how many of the top-ranked alarms we should inspect. One alternative is to stop inspecting alarms when the probability of an alarm drops below some threshold value. Another alternative is to stop when the budget for alarm inspection is exhausted, which could happen in practical scenarios. To guarantee soundness, all the alarms still need to be inspected. In spite of this, our tool will be valuable in practice.

Quality of ranking. As we have seen in the earlier sections, the ranking produced in each interaction of Bingo, and by Presto, ranks alarms from the most highly probable to the least. We enquire into the quality of ranking with the following question: How good is an ordering of alarms, $w = a_1, a_2, \ldots, a_n$, in light of their associated ground truths, v_1, v_2, $\ldots$, v_n?

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

Assume now that $\Pr(\cdot)$ describes the joint probability distribution of alarms. We seek the ordering of alarms with lowest expected inversion count. The following theorem states that the ranking produced by Bingo and Presto is optimal for alarm ranking given a fixed set of observations $\boldsymbol{e}$.

Theorem 4.5.1. *For each set of observations $\boldsymbol{e}$, the sequence $w = a_1, a_2, \ldots, a_n$ of alarms, arranged according to decreasing $\Pr(a_i \mid \boldsymbol{e})$, has the minimum expected inversion count over all potential orderings w'.*

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

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

$$\mathrm{E}(\chi(w) \mid \boldsymbol{e}) = \sum_i \sum_{j>i} \Pr(\neg a_i \wedge a_j \mid \boldsymbol{e}). \tag{4.2}$$

We can now prove the above theorem.

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

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

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

respectively. Our hypothesis states that

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

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

$$\begin{aligned}&\Pr(a_1 \wedge a_2 \mid \boldsymbol{e}) + \Pr(a_1 \wedge \neg a_2 \mid \boldsymbol{e}) \\ &\qquad \geq \Pr(a_1 \wedge a_2 \mid \boldsymbol{e}) + \Pr(\neg a_1 \wedge a_2 \mid \boldsymbol{e}),\end{aligned}$$

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

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

$$\mathrm{E}(\chi(w' \mid \boldsymbol{e})) < \mathrm{E}(\chi(w) \mid \boldsymbol{e}). \tag{4.3}$$

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

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

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

From Equation 4.2 and by an argument similar to that used in the $n = 2$ case, we conclude that $\mathrm{E}(\chi(w^{(k)}) \mid \boldsymbol{e}) \geq \mathrm{E}(\chi(w^{(k+1)}) \mid \boldsymbol{e})$, and transitively that $\mathrm{E}(\chi(w^{(0)}) \mid \boldsymbol{e}) \geq \mathrm{E}(\chi(w^{(m)}) \mid \boldsymbol{e})$. Since $w' = w^{(0)}$ and $w^{(m)} = w$, this immediately conflicts with our assumption that $\mathrm{E}(\chi(w' \mid \boldsymbol{e})) < \mathrm{E}(\chi(w) \mid \boldsymbol{e})$, and completes the proof. $\square$

Another version of this problem which is relevant for Bingo is the alarm ranking in the interactive setting, where $\boldsymbol{e}$ grows with each iteration. This is referring to a "dynamic" ranking of alarms i.e., the sequence of alarms proposed by the interaction model of Bingo to the user. Each alarm in this sequence is the top-ranked one in the alarm ranking produced in each interaction. By choosing a top-ranked alarm during each interaction, Bingo is employing a greedy heuristic. A different strategy for choosing an alarm for inspection during each interaction may potentially yield a better "dynamic" ranking. But finding an optimal strategy in such a setting is a significantly harder problem and one that we do not explore here.

Chapter 5

A Study of Cycle Elimination

In this chapter, we discuss three approaches to eliminate cycles from the derivation graph. Let GC be the set of instantiated constraints comprising the derivation graph. We wish to capture as many probabilistic dependencies of the alarm tuples as possible. One heuristic approach to do this is to construct the Bayesian network from the largest set of clauses $GC_c \subseteq GC$ that induces an acyclic derivation graph. By largest, we mean that the cardinality of GC_c should be as high as possible. Finding the largest subset of acyclic clauses GC_c can be shown to be NP-complete by reduction from the *maximum acyclic subgraph* problem [27]. Therefore, we propose three approaches that relax the "maximum" condition to different extents. We still require every tuple that was derivable in GC to be derivable in GC_c.

The first approach (Section 5.2) constructs GC_c by aggressively removing every clause from GC that derives a tuple that has already been derived "earlier". The second approach (Section 5.3) tries to put each clause discarded by the aggressive algorithm, back into GC_c as long as GC_c induces an acyclic graph. The order in which the discarded clauses are put back into GC_c affects the cardinality of GC_c. Section 5.3 discusses this issue in more detail. The third approach(Section 5.4) transforms GC into a set of clauses GC_{dtcov} that induces an acyclic graph, while retaining more derivation trees for each tuple than the previous two approaches. Intuitively, for each clause discarded by the second approach, it inserts one or more new clauses to retain the derivation trees that comprise the discarded clause, while maintaining acyclicity.

5.1 Notation and Definitions

This section recapitulates all the notation we have used thus far, and states a few definitions. We will use this notation in the following sections.

I :	The set of all input tuples.
C :	The set of all derived tuples.
T :	The set of all tuples. $T = I \cup C$.
GC :	The set of all grounded constraints.
c_g :	The consequent tuple of the grounded constraint $g \in GC$.
A_g :	The set of all antecedent tuples of the grounded constraint $g \in GC$.
$\mathcal{G}(T, GC)$:	The derivation graph induced by the set of tuples T and the set of grounded constraints GC.

Definition 5.1.1 (Tuples of a set of clauses)**.** $tuples(g) = A_g \cup \{c_g\}$ where g is a grounded constraint. $tuples(GC') = \cup_{g \in GC'} tuples(g)$ where GC' is a set of grounded constraints.

Definition 5.1.2 (Cycle in a set of clauses)**.** A sequence of grounded constraints $g_1, g_2, \ldots, g_n, \forall g_i \in GC'$ where $1 \leq i \leq n$ and GC' is any set of clauses, forms a cycle if (a) $c_{g_i} \in A_{g_{i+1}}$ where $1 \leq i < n$, and (b) $c_{g_n} \in A_{g_1}$.

5.2 Aggressive Cycle Elimination

The aggressive cycle elimination algorithm shown in Algorithm 4 is a modified version of the naive Datalog evaluator. For each tuple, it assigns an integer time stamp that captures the number of derivation steps required to derive it, starting from the input tuples. The input tuples are assigned a time stamp of zero. All derived tuples derivable from only input tuples, get a time stamp of one, and so on. A clause in GC that derives a tuple that has already been derived, is not included in GC_c. Note that it is possible for more than one clause to derive a tuple not derived earlier. This property ensures the acyclicity of the derivation graph induced by GC_c.

Algorithm 4 CycleElimAggressive(I, C, GC), where I is the set of all input tuples, C is the set of all derived tuples, and GC is a set of grounded constraints. It returns $GC_c \subseteq GC$, where the set of clauses GC_c induces an acyclic derivation graph.

1. Initialize the timestamp map TS where $TS\colon (I \cup C) \to \mathbb{N}^\infty$, such that for each tuple t, if $t \in I$, $TS(t) = 0$, and otherwise, $TS(t) = \infty$.
2. While there exists a clause g such that

$$\begin{aligned} &TS(c_g) > \max_{a \in A_g}(TS(a)) + 1, \text{ update:} \\ &TS(c_g) \coloneqq \max_{a \in A_g}(TS(a)) + 1. \end{aligned} \tag{5.1}$$

3. Define $GC_c = \{g \in GC \mid TS(c_g) > \max_{a \in A_g}(TS(a))\}$. $GC_c \subseteq GC$ is the set of all those clauses in GC whose consequent has a timestamp *strictly greater* than all of its antecedents.
4. Return GC_c.

Theorem 5.2.1. *For all I, C and GC, if $GC_c =$ CycleElimAggressive(I, C, GC), then (a) $GC_c \subseteq GC$, (b) every tuple derivable using GC is also derivable using GC_c, and (c) GC_c is acyclic.*

Proof. The first part of the claim is immediate because of the definition of GC_c in Algorithm 4.

We will now prove the second part of the claim. If t is an input tuple, then the result is immediate, because of the presence of the clause True $\implies t$ in GC_c. We now consider the derivability of tuple t, which is not an input tuple. At the fixpoint of step 2 of the algorithm, every derivable tuple t has a finite-valued timestamp,

$TS(t) \lneq \infty$. We prove the derivability of t in GC_c by induction on its timestamp $TS(t)$. We already know the result for the case of $TS(t) = 0$, as t is then an input tuple. If $TS(t) = n + 1$, and assuming the result for all tuples t' with $TS(t') \leq n$, consider the deriving clause g_t, and observe that each of its antecedents $t'' \in A_{g_t}$ has a timestamp, $TS(t'') \leq n$. By the induction hypothesis, all these tuples t'' are derivable, and it follows that t is itself derivable within GC_c.

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

5.3 DFS-based Cycle Elimination

In this section we propose a DFS-based approach as shown in Algorithm 5, to eliminate cycles from GC, the set of all grounded constraints. The algorithm initially executes steps 1 to 3 in the same manner as Algorithm 4 to compute the set of "forward" clauses GC_{fwd}, and the set of "backward" clauses GC_{bkwd}. Next, it initializes the acyclic set of clauses GC_c to the set of forward clauses. Then, in steps 6 and 7, it tries to add each clause from the set of backward clauses to GC_c, if the addition of the clause maintains the acyclicity of GC_c. The acyclicity check in step 7(b) executes the standard DFS-based cycle detection algorithm[19] for directed graphs. The correctness of algorithm 5 immediately follows from the theorem below.

Theorem 5.3.1. *A set of clauses GC' contains a cycle iff the directed graph $G = (V, E)$ constructed from GC' with $V = tuples(GC')$ and $E = \{(v_1, v_2) \mid v_1 \in A_g \wedge v_2 = c_g \text{ where } g \in GC'\}$, contains a cycle.*

Proof. Let GC' be some set of clauses containing a cycle. Then by definition, there is a sequence of clauses $g_1, \ldots, g_n$ such that $c_{g_i} \in A_{g_{i+1}}$ where $1 \leq i < n$, and $c_{g_n} \in A_{g_1}$. By the rules of construction for the directed graph $G = (V, E)$, there is a directed edge from c_{g_n} to c_{g_1}, c_{g_1} to c_{g_2}, and so on up to $c_{g_{n-1}}$ to c_{g_n}. These edges form a cycle in $G = (V, E)$.

If there is a directed cycle in graph $G = (V, E)$, then we know that there is a path with directed edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)$ where $1 < n < |V|$. By the rules of construction for graph $G = (V, E)$, for every directed edge (v_i, v_j) in this path, there is at least one grounded constraint $g_i \in GC'$ where tuple $v_i \in A_{g_i}$ and tuple $v_j = c_{g_i}$. The sequence of grounded constraints $g_1, g_2, \ldots, g_n$ form a cycle in GC'. $\square$

However, it is possible that $|GC_c|$ might vary depending on the order in which the clauses in GC_{bkwd} are processed in steps 6 and 7. We can have variants of this algorithm, each with a different heuristic to compute the order in which the clauses in GC_{bkwd} are processed. But our focus will be to explore ways to retain the derivation trees that affect the computation of the probability of a tuple in the resulting Bayesian network (discussed in Section 5.4).

5.4 DT-covering Cycle Elimination

The problem with the previous two approaches is that we lose derivation trees when we discard clauses to make the set of all clauses GC acyclic. If we lose a derivation tree for tuple $t \in T$ (the set of all tuples), then the computation of $\Pr(t)$ (that is, the probability that tuple t is derivable) is more approximate than if we did not lose it. Intuitively, the reason is that a tuple t is not derivable by the analysis only if *none* of the derivation trees for t derive it. This means that we need to track all the derivation trees for a tuple t in order to determine the probability of tuple t being derivable. We first formally define the probability $\Pr(t)$ of a tuple t being derivable, as the probability of an event that contains all possible outcomes in a probability space, in which the tuple t is derivable. Next, we express $\Pr(t)$ for a tuple t in terms of the derivation trees that derive it.

Algorithm 5 CycleElimDfs(I, C, GC), where I is the set of all input tuples, C is the set of all derived tuples, and GC is a set of grounded constraints. It returns $GC_c \subseteq GC$, where the set of clauses GC_c induces an acyclic derivation graph.

1. Initialize the timestamp map TS where $TS\colon (I \cup C) \to \mathbb{N}^\infty$, such that for each tuple t, if $t \in I$, $TS(t) = 0$, and otherwise, $TS(t) = \infty$.
2. While there exists a clause g such that

$$\begin{aligned} &TS(c_g) > \max_{a \in A_g}(TS(a)) + 1, \text{ update:} \\ &TS(c_g) \coloneqq \max_{a \in A_g}(TS(a)) + 1. \end{aligned} \tag{5.2}$$

3. Define $GC_{fwd} = \{g \in GC \mid TS(c_g) > \max_{a \in A_g}(TS(a))\}$. $GC_{fwd} \subseteq GC$ is the set of all those clauses in GC whose consequent has a timestamp *strictly greater* than all of its antecedents.
4. Define $GC_{bkwd} = \{g \in GC \mid TS(c_g) \leq TS(a), \text{for some } a \in A_g\}$. $GC_{bkwd} \subseteq GC$ is the set of all those clauses in GC whose consequent has a timestamp less than or equal to that of at least one of its antecedents.
5. Initialize $GC_c = GC_{fwd}$.
6. Construct a directed graph $G = (V, E)$ from GC_{fwd} with

$$\begin{aligned} V &= \mathit{tuples}(GC_{fwd}), \text{and} \\ E &= \{(v_1, v_2) \mid v_1 \in A_g \wedge v_2 = c_g \text{where } g \in GC_{fwd}\}. \end{aligned}$$

7. For each clause $g \in GC_{bkwd}$:
   (a) Construct a directed graph $G' = (V', E')$ where $V' = V \cup \mathit{tuples}(g)$ and $E' = E \cup \{(a, c_g) \mid a \in A_g\}$.
   (b) If IsAcyclic(G'):
      i. Update $G = G'$.
      ii. Update $GC_c = GC_c \cup \{g\}$.
8. Return GC_c.

As we have seen before, the computation model to compute $\Pr(t)$ for a tuple t is a Bayesian network extracted from the derivation graph induced by the set of clauses GC. To be able to extract a Bayesian network, we need to eliminate cycles from GC. We consider the problem of not losing derivation trees for a tuple $t \in T$. In general, a tuple might have infinite derivation trees. So we define for each tuple $t \in T$, the notion of a finite set of *contributing* derivation trees that is equal to or a subset of all possible derivation trees for tuple t. We show that the contributing set of derivation trees for tuple t is sufficient to compute the probability $\Pr(t)$ as defined in Section 5.4.1. Based on this theory, we derive a technique to eliminate cycles while preserving at least the contributing derivation trees, for a tuple $t \in T$.

This approach entails a *transformation* of the set of clauses GC to another set of clauses GC_{dtcov}. This transformation should ensure that (a) all the tuples derivable in GC are derivable in GC_{dtcov} (b) GC_{dtcov} induces an acyclic derivation graph, and (c) for each tuple $t \in T$, at least a contributing set of derivation trees in GC, is retained in GC_{dtcov}.

Section 5.4.1 builds up the necessary theory and Section 5.4.2 presents the algorithm that achieves the transformation of GC to GC_{dtcov} as specified above.

5.4.1 Underlying Theory

When an analysis is applied to a program, the analysis conclusions are incomplete: not all conclusions hold true in the ground truth. Since we have no way of knowing the ground truth, we consider a probabilistic model associated with the analysis such that given probabilistic assumptions about the input to the analysis and the rules of the analysis, we can infer corresponding probabilities for the analysis conclusions.

Our approach is inspired by probabilistic databases [76]. We postulate that the presence of input tuples and of the grounded instances of the constraints is associated with a probability and that these events are mutually independent. In particular, for each constraint r there is a number w_r where $0 \leq w_r \leq 1$, such that the use in execution of every grounded instance of the constraint has the probability w_r. If there are n grounded constraints (including input tuples) in GC, then our probability space has 2^n possible worlds or outcomes. In each world, each grounded constraint $g \in GC$ is either in (in other words, it "fires") or out. Therefore, the probability of each world W is $\prod_{g \in W} w_r \cdot \prod_{g \notin W}(1 - w_r)$ where r is the constraint corresponding to the grounded constraint g.

Definition 5.4.1 (Probability Space)**.** Our probability space is the pair (Ω, P) where the sample space Ω is the set of all possible outcomes. It is the set of subsets of GC where each subset represents the grounded constraints reached by the execution of the Datalog analysis. If n is the number of grounded constraints, then $\mid \Omega \mid = 2^n$. The function $P : \Omega \rightarrow [0, 1]$ assigns probabilities to each outcome.

Definition 5.4.2 (Derivation Tree)**.** A derivation tree τ for a tuple t in T is a labelled tree in which: (a) each vertex of the tree is labelled by one tuple, (b) the root is labelled by t, (c) the leaves are labelled by tuples in I, (d) each internal vertex is labelled by tuple t_h and its children are respectively labelled by tuples $t_1, t_2, \ldots, t_n$ if the grounded constraint $t_1 \wedge t_2 \wedge \ldots \wedge t_n \implies t_h \in GC$. We say that a node of a derivation tree represents a grounded constraint g if it is labelled by the consequent tuple of g, and its children are labelled by the antecedent tuples of g. We define $nodes(\tau)$ to be the set of nodes of the derivation tree τ.

Definition 5.4.3 (Clauses of a Derivation Tree)**.** Let τ be a derivation tree. Define $gc(\tau) = \{g \mid g \in GC$, g is of the form $t_1 \wedge t_2 \wedge \ldots \wedge t_n \implies t_h$, and τ has an internal node labelled t_h with its children labelled t_1 through $t_n\}$. Thus, $gc(\tau)$ is the set of all grounded constraints used in the derivation tree τ.

We are interested in the following kinds of events over the sample space Ω:

1. k_g: The event that comprises all outcomes from the sample space Ω in which the grounded constraint g fires. We define $P(k_g) = w_r$.

2. k_τ: The event that comprises all outcomes from the sample space Ω in which the derivation tree τ "fires". For the derivation tree τ to fire, all the grounded constraints from $gc(\tau)$ must fire. Therefore, $k_\tau = \bigcap_{g \in gc(\tau)} k_g$ and $P(k_\tau) = \prod_{g \in gc(\tau)} P(k_g)$.

3. k_t: The event that comprises all outcomes from the sample space Ω in which tuple t is derivable. A tuple t is derivable if at least one derivation tree deriving it, fires. Therefore, $k_t = \bigcup_{\tau \text{ derives } t} \bigcap_{g \in gc(\tau)} k_g$. We need to compute $P(k_t)$.

With each of the above events, we associate a boolean random variable. Let the random variable X_g represent the event k_g, the random variable X_τ represent the event k_τ, and the random variable X_t represent the event k_t. Then:

$$X_g : \Omega \to \{true, false\} \text{ where } \forall o, X_g(o) = true \text{ if } o \in k_g, false \text{ otherwise.}$$
$$X_\tau : \Omega \to \{true, false\} \text{ where } \forall o, X_\tau(o) = true \text{ if } o \in k_\tau, false \text{ otherwise.}$$
$$X_t : \Omega \to \{true, false\} \text{ where } \forall o, X_t(o) = true \text{ if } o \in k_t, false \text{ otherwise.}$$

The shorthand notation for $P(\{o \mid X_t(o) = true\})$ is $\Pr(t)$. We similarly define $\Pr(g)$ and $\Pr(\tau)$.

Next, we define the contributing set of derivation trees that we have motivated earlier.

Definition 5.4.4 (Contributing Set of Derivation Trees)**.** A set of derivation trees $R = \{\tau_1, \tau_2, \ldots, \tau_n\}$ for a tuple $t \in T$ is said to be a contributing set if

$$\text{(Incomparability) } \forall \tau, \tau' \in R,\ gc(\tau) \nsubseteq gc(\tau'),$$
$$\text{(Exhaustivity) } \forall \tau \text{ s.t. } \tau \text{ derives } t,\ \tau \notin R \implies \exists \tau' \in R \text{ s.t. } gc(\tau') \subseteq gc(\tau), \text{ and}$$
$$\text{(Minimality) } \forall \tau \text{ s.t. } \tau \text{ derives } t,\ \tau \notin R \implies \exists \tau' \in R \text{ s.t. } gc(\tau') = gc(\tau) \implies |\,nodes(\tau')\,| \leq |\,nodes(\tau)\,|\ .$$

Lemma 5.4.1. *A contributing set of derivation trees for a tuple t is finite.*

Proof. It is clear from the definition of the contributing set of derivation trees that for any pair of trees τ and τ' belonging to it, $gc(\tau) \neq gc(\tau')$. Since the set of grounded constraints GC is finite, the number of distinct subsets of GC is finite and equal to $2^{|GC|}$. The size of a contributing set is at most $2^{|GC|}$. □

The lemma below states that the set of outcomes from the sample space Ω, represented by the contributing derivation trees for a tuple t, contains all outcomes from Ω in which tuple t is derivable. Therefore, retaining at least the contributing derivation trees from the set of clauses GC, in the transformed set of clauses GC_{dtcov} is sufficient to compute $\Pr(t)$.

Lemma 5.4.2. *For any tuple t, let R be its contributing set of derivation trees. Then, $\bigcup_{\tau \text{ derives } t} k_\tau = \bigcup_{\tau \in R} k_\tau$.*

Proof. Let $R = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a contributing set of derivation trees for tuple t. Let τ be a derivation tree for tuple t s.t. $\tau \notin R$. Then by definition of the contributing set, $\exists \tau' \in R$ s.t. $gc(\tau') \subseteq gc(\tau)$. Now, $k_{\tau'}$ is the set of all outcomes from the sample space Ω in which the grounded constraints in $gc(\tau')$ have fired. Since $gc(\tau)$ contains all the grounded constraints of τ' and maybe more, the outcomes in which all of $gc(\tau)$ fire will be a subset of $k_{\tau'}$. That is, $k_{\tau'} \supseteq k_\tau$. Therefore, $k_{\tau'} \cup k_\tau = k_{\tau'}$. Since this holds for any derivation tree τ for tuple t that is not in R, the lemma follows. □

The lemma below formally states and proves the following statement: if a derivation tree for tuple t uses all the clauses forming a cycle along one of its paths, then it cannot be a contributing derivation tree for t.

Lemma 5.4.3. *Let $g_1, g_2, \ldots, g_m$ be a sequence of clauses that forms a cycle, where each $g_i, 1 \leq i \leq m$ belongs to the set of grounded constraints GC. A derivation tree τ for tuple t, that has a path p with a sequence of nodes that successively represent $g_i, g_{i-1}, g_{i-2}, \ldots, g_1, g_m, g_{m-1}, \ldots, g_{i+1}$ where $1 \leq i \leq m$, is not a contributing derivation tree for tuple t.*

Proof. Let the sequence of nodes on path p that represent clauses $g_i, g_{i-1}, g_{i-2}, \dots, g_1$, $g_m, g_{m-1}, \dots, g_{i+1}$, be $n_i, n_{i-1}, n_{i-2}, \dots, n_1, n_m, n_{m-1}, \dots, n_{i+1}$. As clauses $g_i, \dots g_{i+1}$ form a cycle, one of the antecedent tuples of the clause g_{i+1} is tuple c_{g_i}. Let n be the node representing this tuple, with s as the subtree rooted at n. We construct a derivation tree τ' from derivation tree τ by deleting the nodes $n_{i-1}, n_{i-2}, \dots, n_1, n_m, n_{m-1}$, $\dots, n_{i+1}$, and the node n on path p. Next, we make the children of the root of subtree s as the children of node n_i in derivation tree τ'. Note that the root of subtree s and node n_i are labelled by the same tuple. Thus, $gc(\tau') \subseteq gc(\tau)$. Both τ and τ' are derivation trees for tuple t. Moreover, we will delete at least one node from derivation tree τ to form derivation tree τ' because at least one clause is required to form a cycle that will result in at least two nodes on path p. Therefore, by the incomparability and minimality conditions, the derivation tree τ can never be a contributing derivation tree for tuple t. $\square$

Next, we develop a succinct representation for all the derivation trees of a tuple in the set of grounded constraints GC. Our representation is motivated by the study of provenance polynomials in databases [31]. This representation is used by the algorithm given in the next section to keep track of all the derivation trees for a tuple. In order to treat input and derived tuples uniformly, we augment GC with one constraint for each input tuple: $g_e : true \implies e$ where $e \in I$.

Let $(K, +, ., 0_K, 1_K)$ be a commutative semiring where each element of K is a set of sets of grounded constraints. Therefore, $\mid K \mid = 2^{2^{|GC|}}$. Let k_1 and k_2 be two elements of K. We define the operations of $+$ and $.$ as follows:

Operator $+$ is the global union, therefore, $k_1 + k_2 = k_1 \cup k_2$.

Operator . is pairwise union, therefore, $k_1.k_2 = \{e_1 \cup e_2 \mid e_1 \in k_1 \wedge e_2 \in k_2\}$

The additive and multiplicative identities are: $0_K = \{\}$ and $1_K = \{\{\}\}$. We denote an element of K as a sum of terms. For example, an element $\{\{g_1, g_2\}, \{g_3, g_4\}, \{g_5\}\}$ of K is denoted as $g_1 g_2 + g_3 g_4 + g_5$.

We use the elements of the commutative semiring K to succinctly represent derivation trees of a tuple, as a sum of terms. Each term represents a derivation tree τ by the set $gc(\tau)$. For example, let a tuple have three derivation trees τ_1, τ_2, and τ_3, where $gc(\tau_1) = \{g_1, g_2\}$, $gc(\tau_2) = \{g_3, g_4\}$, and $gc(\tau_3) = \{g_5\}$. Then the element of K that represents all the above three derivation trees, is $g_1g_2 + g_3g_4 + g_5$. This representation is succinct because the operations of the semiring K fuse the representations of two distinct derivation trees into one, if they comprise the same set of clauses.

Next, we define the notion of a polynomial over the semiring K. We need this to represent the derivation tree of a tuple in terms of derivation trees of other tuples. A polynomial in our setting, is an expression over a set of variables $X = \{x_{t_1}, x_{t_2}, \ldots\}$. Each variable x_{t_i} that may occur in a polynomial, is associated with the tuple $t_i \in T$, and represents all the derivation trees for the tuple t_i. Therefore, there are $\mid T \mid$ variables, one for each tuple in T, where T is the set of all tuples. Now, we can express a derivation tree for a tuple t as a polynomial over the variables corresponding to the antecedent tuples of t, in that derivation tree. We illustrate with an example. Let g_1: $t_1 \wedge t_2 \implies t$ and g_2: $t_3 \implies t$ be two ways in which we can derive tuple t. Here, clauses $g_1, g_2 \in GC$, tuple $t \in C$, and tuples $t_1, t_2, t_3 \in T$. We represent the derivation trees for tuple t by the variable x_t, for tuple t_1 by variable x_{t_1} and so on. The polynomial for x_t is expressed in terms of variables x_{t_1}, x_{t_2} and x_{t_3} as follows: $x_t = g_1x_{t_1}x_{t_2} + g_2x_{t_3}$. We say that the term $g_1x_{t_1}x_{t_2}$ *represents* the grounded constraint g_1, the term $g_2x_{t_3}$ *represents* the grounded constraint g_2, and the polynomial expression $g_1x_{t_1}x_{t_2} + g_2x_{t_3}$ *represents* the grounded constraints that derive tuple t. Note that if a term in a polynomial expression has variables, then it represents multiple derivation trees each comprising a distinct set of clauses.

Definition 5.4.5 (Terms of a Polynomial Expression)**.** Let *expr* be a polynomial expression over the semiring K. Define $terms(expr) = \{trm \mid trm$ is a term in the polynomial expression $expr\}$.

Definition 5.4.6 (Substitution in a Polynomial Expression)**.** For any tuple $t \in T$, let $expr_t$ denote the polynomial expression that represents the grounded constraints that derive tuple t. Let pp be any polynomial expression, one of whose terms refers to the variable x_t for some tuple t. Let $newpp$ be the polynomial expression got by substituting $expr_t$ for variable x_t in pp. We denote this action of substituting for all occurrences of one variable in pp, as $pp \Rightarrow newpp$. The action of substituting for zero or more variables, one after the other in sequence, is denoted by $pp \Rightarrow^* newpp'$, where $newpp'$ is a polynomial expression.

Thus, performing one or more substitutions on a polynomial expression that represents all the derivation trees of a tuple t, will yield us a different polynomial expression that still represents all the derivation trees of tuple t. Therefore, the general form of a term is: $g_1 \ldots g_n x_{t_1} \ldots x_{t_m}$. Just as we illustrated how a term of a polynomial can be constructed from a clause, we define the inverse operation of extracting a clause from a term of a polynomial expression that represents the derivation trees of a tuple.

Definition 5.4.7 (Clause Corresponding to a Term in a Polynomial Expression)**.** Let trm be a term in the polynomial expression that represents the grounded constraints that derive tuple t, for any tuple $t \in T$. Define $get\_clause(trm) = t_1 \wedge \ldots \wedge t_m \implies t$ where $trm = g_1 \ldots g_n x_{t_1} \ldots x_{t_m}$.

Lemma 5.4.4. *For any tuple $t \in T$, let $expr_t$ denote the polynomial expression that represents the grounded constraints that derive tuple t. If $expr_t \Rightarrow^* expr'_t$, and there is a term in $expr'_t$ that refers to the variable x_t, then that term represents the derivation trees for tuple t that have a cycle in one of their paths.*

Proof. Let $expr_t$ denote the polynomial that represents the grounded constraints that derive tuple t.

Case 1. Let $expr_t \Rightarrow^* expr'_t$ with zero substitutions, so $expr_t = expr'_t$. Let $expr_t$ contain the term $g. \ldots .x_t. \ldots$, where the grounded constraint $g \in GC$. Then, clause

g has the form $\ldots \wedge t \wedge \ldots \implies t$. This term represents derivation trees for tuple t whose root node is labelled by tuple t. The child nodes of the root node will be labelled by tuples in A_g, one of which is tuple t. The path from the root node to the child node labelled by tuple t, in each of these derivation trees, represents the cycle formed by the clause g.

Case 2. Let $expr_t \Rightarrow^* expr'_t$ with one or more substitutions. Let the sequence $expr^1_t, \ldots, expr^n_t$ be the polynomial expressions after each substitution, with $expr_t = expr^1_t$ and $expr^n_t = expr'_t$. By definition of the substitution operation, for each $1 \leq i < n$, $expr^{i+1}_t$ is got by substituting for some variable x_{t_i} occurring in a term of $expr^i_t$. Again, by definition, variable x_{t_i} is substituted by the polynomial expression that represents the grounded constraints that derive tuple t_i. We know that $expr^n_t$ contains some term that refers to variable x_t. So, there must be a sequence of clauses $g_1, \ldots, g_n$ such that tuple $t_i = c_{g_i}$ and tuple $t_{i+1} \in A_{g_i}$, for $1 \leq i < n$, and where tuple $t_n = t = c_{g_n}$. Also, there must be some clause $g \in GC$ where tuple $t = c_g$ and tuple $t_1 \in A_g$ because $expr^1_t$ is the polynomial expression that represents the grounded constraints that derive tuple t. Therefore, the sequence of clauses $g, g_1, \ldots, g_n$ form a cycle. Let $term_g$ be the term in $expr^1_t$ that represents the grounded constraint g. Since the derivation tree has an edge from a node representing a consequent tuple to a node representing an antecedent tuple, the derivation trees represented by $term_g$ have a path that represents the cycle caused by clauses $g, g_1, \ldots, g_n$. $\square$

Definition 5.4.8 (DFS Descendants)**.** Let the set of clauses GC induce the directed graph $G = (V, E)$ where $V = tuples(GC)$ and $E = \{(v_1, v_2) \mid v_1 \in A_g \wedge v_2 = c_g \text{ where } g \in GC\}$. Further, let ds be the set of trees resulting from a depth first traversal of graph G. For any tuple $t \in tuples(GC)$, define $dfs\_descendants(t) = \{t' \mid t' \text{ is a vertex in the subtree rooted at } t, \text{ within some tree in } ds\}$.

We note that the depth first traversal of a graph, initiated at different sets of root nodes, may yield different sets of depth first trees. We have elided this level of detail in order to keep the notation simple. The above definition of $dfs\_descendants$ is with respect to a given depth first traversal.

Note that if a sequence of grounded constraints $g_1, g_2, \ldots, g_n \in GC$ form a cycle, and if some tuple t is the earliest tuple among $tuples(\{g_1, \ldots, g_n\})$ to be visited by a depth first traversal of graph G induced by GC, then by the property of depth first traversal, $tuples(\{g_1, \ldots, g_n\}) \setminus \{t\} \subseteq dfs\_descendants(t)$. Moreover, if a different tuple $t' \in tuples(\{g_1, \ldots, g_n\})$ happens to be the earliest tuple visited by a different depth first traversal, then all the remaining tuples $(tuples(\{g_1, \ldots, g_n\}) \setminus \{t'\})$ will still belong to $dfs\_descendants(t')$ as determined by the different depth first traversal.

Lemma 5.4.5. *For each tuple $t \in T$, let $expr_t$ denote the polynomial expression that represents the derivation trees of tuple t. Let $dfs\_descendants(t)$ be the DFS descendants in a depth first traversal of the graph G induced by the set of clauses GC. For each tuple $t \in T$, let $expr_t \Rightarrow^* expr'_t$ s.t. all the terms in $expr'_t$ do not refer to a variable that corresponds to any tuple $\in dfs\_descendants(t)$. Define $GC' = \{g \mid g = get\_clause(trm) \;\; where \;\; trm \in terms(expr'_t) \wedge trm$ does not refer to x_t, $\forall t \in T\}$. Then, GC' is acyclic.*

Proof. To see this, consider any term trm, which contributes a clause to GC', and which occurs in the polynomial expression $expr'_t$ for some tuple t. Let $x_{t'}$ be a variable referred to in trm. By the conditions stated in the Lemma, $t' \neq t$ and $t' \notin dfs\_descendants(t)$. It is evident from the rules of construction of a polynomial expression for a tuple, and from the definition of $\Rightarrow^*$ that tuple t' derives tuple t in GC'. We show that GC' is acyclic by showing that tuple t cannot derive tuple t' in GC'. We prove this by contradiction.

Assume that tuple t derives tuple t'. We know that $t' \notin dfs\_descendants(t)$. This implies that t' must have been visited earlier than t in the depth first traversal. Putting this fact together with the fact that t' derives t, we conclude that t' must be an ancestor of t in the depth first traversal. In other words, $t \in dfs\_descendants(t')$. By the pre-conditions of the lemma, there will be no term in the polynomial expression for tuple t' in which variable x_t will occur. A similar argument as above leads us to conclude that t must be an ancestor of t', which leads to a contradiction. Therefore, it is not possible that tuple t derives tuple t', which implies that there is no cycle involving tuples t and t'. □

Algorithm 6 CYCLEELIMDTCOVERING(GC, W, I, C), where GC is the set of clauses and W is a function that associates each clause to its probability. I and C are the sets of input and derived tuples. It returns GC_{dtcov}, the new set of clauses got by transforming GC, and the probabilities W_{dtcov} for clauses in GC_{dtcov}.

```
1. Initialize $PP(t) = 0_K,\ \forall t \in (I \cup C)$ where $PP : C \rightarrow$ polynomials over $K$.
2. For each grounded constraint $g : t_1 \wedge \ldots \wedge t_n \implies t_h \in GC$, update:
      $$PP(t_h) = PP(t_h) + g.x_{t_1}.x_{t_2}. \ \ldots \ .x_{t_n}. \quad (5.3)$$
3. Initialize $roots = \{c_g | A_g \subseteq I \wedge g \in GC\}$. The set $roots$ contains the tuples that
   are directly derivable from input tuples.
4. Initialize $visited(t) = false,\ \forall t \in C$ where $visited : C \rightarrow \{true, false\}$.
5. While there is a tuple $t \in roots$ such that $visited(t) = false$, call UPDATEPP$(t)$.
6. Initialize $GC_{dtcov} = \{\}$. $GC_{dtcov}$ is the new set of grounded constraints that is
   populated in the next step.
7. For each tuple $t \in C$, do:
      For each term $g_1.g_2. \ \ldots \ .g_n.x_{t_1}. \ \ldots \ .x_{t_k}$ in $PP(t)$, execute the following steps.
         a. Update $GC_{dtcov} = GC_{dtcov} \cup g'$ where $g' : t_1 \wedge \ldots \wedge t_k \implies t$. In effect,
            $g'$ is the constraint got by merging constraints $g_1, g_2, \ldots, g_n$.
         b. Update $W_{dtcov}(g') = W(g_1).W(g_2). \ \ldots \ .W(g_n)$ where $W_{dtcov} :$
            $GC_{dtcov} \rightarrow [0, 1]$. The probability of $g'$ is the product of the probabili-
            ties of $g_1, \ldots, g_n$.
8. Return $GC_{dtcov}$ and $W_{dtcov}$.
```

Algorithm 7 UPDATEPP(t) where t is the tuple being processed. It returns *descendants*, the set of all tuples that are DFS descendants of tuple t. Variables GC (the set of all grounded constraints), I (the set of input tuples), C (the set of derived tuples), PP (a function from each tuple to its polynomial), and *visited* (a function that maps each tuple to whether it is processed or not) are assumed to be available globally.

```
1. Update $visited(t) = true$.
2. Initialize $descendants = \{\}$. The set $descendants$ collects all tuples that are DFS
   descendants of tuple $t$.
3. Let $neighbours = \{c_g \mid t \in A_g \wedge g \in GC\}$. This is the set of all tuples derivable
   from tuple $t$ in one step.
4. For each tuple $t' \in neighbours$ and $visited(t') = false$, do:
    (a) Let $descendants' = UpdatePP(t')$. The recursive call updates the polyno-
        mial for tuple $t'$. The set $descendants'$ contains the tuples that are DFS
        descendants of tuple $t'$.
    (b) Update $descendants = descendants \cup \{t'\} \cup descendants'$.
5. Let $expr = PP(t)$. At this point, the polynomials of all tuples transitively derivable
   from tuple $t$ are updated to their final form.
6. Remove from $expr$ all terms in which variable $x_t$ occurs. A term in which variable
   $x_t$ occurs represents derivation trees of tuple $t$ that have a cycle.
7. While there is a tuple $t'' \in descendants$ such that variable $x_{t''}$ occurs in a term of
   $expr$, do:
    (a) Substitute every occurrence of variable $x_{t''}$ with polynomial $PP(t'')$, in $expr$.
    (b) Remove from $expr$ all terms in which $x_t$ occurs.
8. Update $PP(t) = expr$. At this point, the polynomial for tuple $t$ is updated to its
   final form. It now refers to only those variables corresponding to tuples that are
   not derivable from tuple $t$.
9. Return $descendants$.
```

5.4.2 Algorithm

This section gives the algorithm (specified in two parts as Algorithms 6 and 7) to transform the set of grounded constraints GC to another set of grounded constraints GC_{dtcov}. At a high level, the algorithm does the following:

1. For each derived tuple, the algorithm records all the derivation trees in the form of a polynomial expression representing the grounded constraints deriving that tuple.

2. It simulates a depth first search on an underlying directed graph $G = (V, E)$ where the set of vertices V represents derived tuples, and there is a directed edge between two vertices v_1 and v_2 if the tuple represented by v_1 derives the tuple represented by v_2 in one step.

3. The key idea of the algorithm is that it recognizes the presence of a cyclic set of grounded constraints when it observes that the polynomial expression of tuple t refers to a variable that corresponds to a DFS descendant of tuple t.

4. It ensures acyclicity of the transformed set of clauses GC_{dtcov}, and preserves the contributing derivation trees by performing the following actions on the polynomial expression of each tuple t.

    (a) It deletes terms that refer to variable x_t (x_t succinctly represents all the derivation trees for tuple t). By Lemmas 5.4.3 and 5.4.4 such terms do not represent contributing derivation trees.

    (b) It transforms the remaining terms until they no longer refer to variables that represent DFS descendants of tuple t. By Lemma 5.4.5, once this transformation is performed on the terms of the polynomial expressions of all the derived tuples, the resulting derivation graph GC_{dtcov} is acyclic.

GC_{dtcov} retains all contributing derivation trees from GC for any tuple t. Lemma 5.4.2 shows that retaining all contributing derivation trees is a sufficient condition to compute $\Pr(t)$ for a tuple t.

Here is a step-by-step description of the algorithm. Algorithm 6 first records the polynomial for each derived tuple t, in $PP(t)$, in step 2. The polynomial for tuple t is in terms of the variables representing its immediate antecedent tuples. Next, it initializes the book-keeping variables of the algorithm: *roots* and *visited*. The variable *visited* is global and is visible to Algorithm 7. In step 5, Algorithm 7 is invoked on a tuple that is derived only from input tuples.

Algorithm 7 recursively visits every tuple that is transitively derivable from its argument, in a depth-first manner. During this depth first visit, Algorithm 7 updates the polynomial $PP(t)$ of its argument tuple t *after* it has completed the recursive calls on all its depth-first descendants. Therefore, when Algorithm 7 updates the polynomial for a tuple t, the polynomials for all tuples that tuple t transitively derives, are already updated. The update step for tuple t entails repeatedly substituting for every occurrence of variable $x_{t'}$ by the polynomial $PP(t')$, in the polynomial for tuple t, where tuple t' is a DFS descendant of tuple t. This substitution continues as long as the polynomial for tuple t refers to variables corresponding to DFS descendants of tuple t, and stops when it refers only to the variables corresponding to the DFS non-descendants of tuple t. At every step of the substitution, if a term in the polynomial $PP(t)$ refers to x_t, that term is discarded as it represents derivation trees for tuple t that have a cycle in one of their paths. After every derived tuple has been visited, Algorithm 6 reads off the new grounded constraints and their probabilities from the updated polynomials for each derived tuple, in steps 6 and 7.

5.5 Illustrative Example

In this section we illustrate the effect of the three cycle elimination algorithms on a small example, and show how they are increasingly more precise. Figure 5.1(a) shows a toy derivation graph that has four clauses g_1, g_2, g_3, g_4.

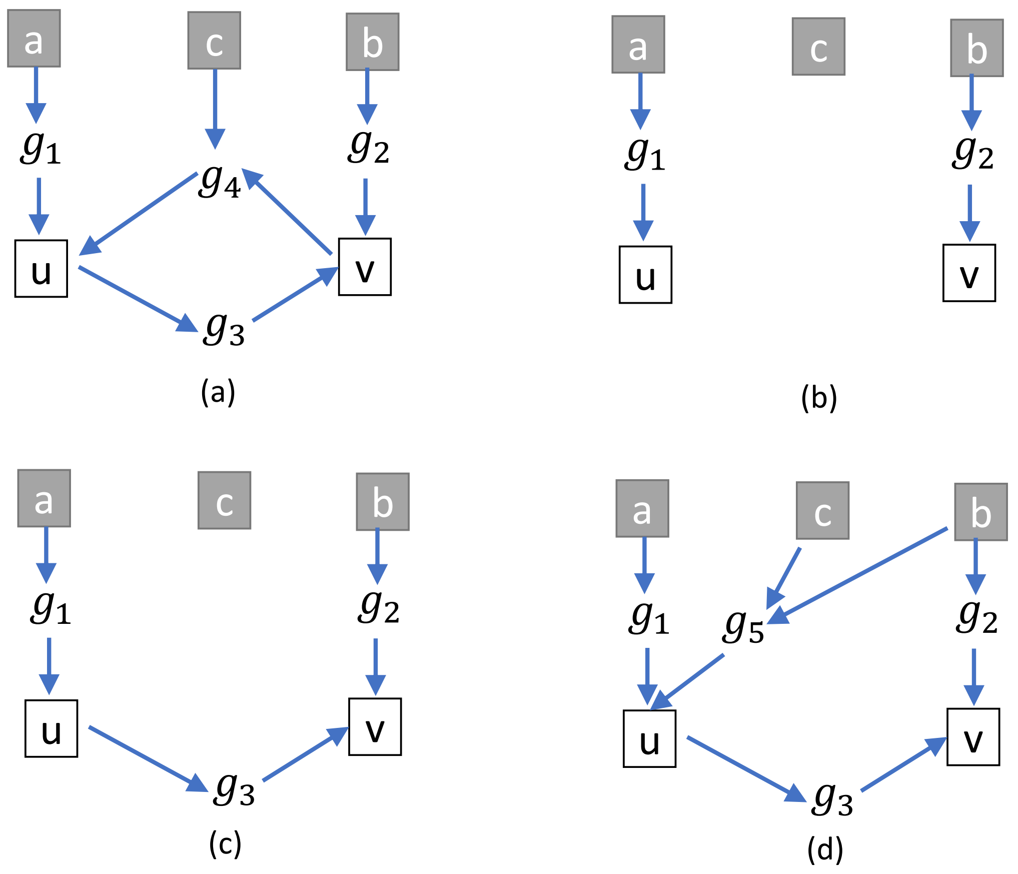


Figure 5.1: An example to illustrate the effect of the three cycle elimination algorithms. (a) is an example graph, (b) is after Aggressive cycle elimination, (c) is after DFS-based cycle elimination, and (d) is after DT-covering cycle elimination. Observe the increasing precision of these algorithms.

Aggressive cycle elimination removes the clauses g_3 and g_4 from the derivation graph to make it cycle-free. A contributing derivation for tuple u ($\{g_2, g_4\}$) is lost. Similarly, a contributing derivation for tuple v ($\{g_1, g_3\}$) is also lost. Figure 5.1(b) shows the derivation graph after aggressive cycle elimination is applied to it.

DFS-based cycle elimination puts the clause g_3 back into the cycle-free derivation graph produced by the aggressive cycle elimination algorithm, because it confirms that adding clause g_3 will maintain the acyclicity of the resulting graph. Therefore, the DFS-based algorithm retains all contributing derivations for tuple v but still loses a contributing derivation for tuple u. Figure 5.1(c) shows the derivation graph after DFS-based cycle elimination is applied to it.

The DT-covering cycle elimination algorithm retains clauses g_1, g_2 and g_3 but recognizes that it cannot retain clause g_4 because it will introduce a cycle. Instead, it introduces a "new" clause g_5. Observe that clause g_5 is actually a compression of clauses g_2 and g_4. Accordingly, the probability associated with clause g_5 is the product of the probabilities associated with clauses g_2 and g_4. Now, a contributing derivation tree $\{g_2, g_4\}$ for tuple u that was previously lost, is substituted by the derivation tree $\{g_5\}$. In this way, the DT-covering cycle elimination algorithm retains all contributing derivations for all tuples of a derivation graph. Figure 5.1(d) shows the derivation graph after DT-covering cycle elimination is applied to it.

Chapter 6

Related Work

There is a large body of research on techniques to overcome the incompleteness of static program analysis tools. Muske et al [62] give a detailed survey of different techniques used by static analyses for alarm handling, their merits and shortcomings. We broadly classify this research into techniques based purely on static reasoning, techniques that interact with users, techniques that interact with dynamic analyses, and techniques that take a statistical approach. We elaborate below on each of these categories.

Techniques based purely on static program reasoning. Lee et al [48] cluster correlated alarms by discovering sound dependencies between them such that if the dominant alarms of a cluster turns out to be false, all the other alarms in the same cluster are guaranteed to be false. Le and Soffa [46], and also Zhang et al [80] define two alarms to be correlated if one alarm causes the other alarm to occur. Le and Soffa [46] first statically detect alarms and determine their error states. Next, they propagate the effects of error states along paths, to automatically detect correlated pairs of alarms. From this, they construct a correlation graph showing correlations among multiple alarms along different paths. Using the correlation graph, they minimize the number of reported alarms required to find the root cause of an alarm.

Kahlon et al [35] statically detect datarace alarms, and track lock acquisition patterns to detect alarms that may potentially be false. Since their technique is unsound, they use it to rank alarms rather than filter them. Sherriff et al [75] also use unsound methods that leverage historical field failure records to group static analysis alarms that are predictive of a field failure. However, all such techniques are constrained by the limits of logical reasoning, and cannot extract fine correlations between alarms (such as, "If a is false, then b is also *likely* to be false.").

Techniques that interact with users. These techniques interact with a user in order to reduce false positives in alarm reports. They either take some specifications provided by a user, or they pose queries to a user, which they identify by reasoning about the logical structure of the analysis and the program in question. Dillig et al [22] formulate a search for missing facts to discharge an alarm as an abductive inference problem. Ivy [66] graphically displays succinct counterexamples to the user to help identify inductive invariants. URSA [81] formulates the search for the root causes of false alarms in Datalog derivation graphs as an information gain maximization problem. While all these systems pose queries to a user, some of these systems pose queries about program facts that are *related* to an alarm, and not about the alarm itself. For example, URSA poses queries on the root cause of an alarm. Depending on the complexity of the static analysis, such queries may place an undue burden on a user. Some techniques classify alarms with a single round of user feedback [55]. But, unlike Bingo, all these techniques do not iteratively maximize the return on a user's effort expended on providing correct feedback. Le and Soffa [47] propose a framework that requires as input a user-provided specification in order to automatically generate scalable, interprocedural, path-sensitive analyses to detect user-specified alarms. This specification needs to express alarms and information needed for their detection, a scalable, path-sensitive algorithm, and a generator that unifies the two. The analysis produced from this specification identifies alarms and also the path segments where the root causes of an alarm are located. Other techniques leverage analysis-derived features, e.g., to assign confidence values to alarms [51], but they rely on expert users to assign weights to analysis rules.

Techniques that interact with dynamic analysis. The goal of such techniques is to reap the benefits of the soundness of static analysis, while offsetting its incompleteness with the completeness of dynamic analysis, or testing. Csallner et al [20] present an automatic error-detection technique that combines static checking and concrete test-case generation. Their approach is to produce concrete test cases by deriving specific error conditions from the abstract error conditions inferred by a static analysis. These test cases are then executed to determine whether an error truly exists. In a later work, Csallner et al [21] propose a 3-step approach to bug-finding. The first step is dynamic invariant detection to capture a program's intended behavior, the second step is to statically analyze the program within the restricted input domain allowed by the detected invariants, and the third step is automatically generate test cases directed by the predictions of the static analysis. They claim higher precision over tools that lack a dynamic step and higher efficiency over tools that lack a static step. In yet another work, Li et al [50] enlist the support of a dynamic technique that automatically validates and categorizes the numerous, but potentially false, memory-leak warnings reported by a static memory-leak detector. Kiss et al [38] also combine static and dynamic analysis techniques to detect vulnerabilities, and illustrate how their tool finds a simplified version of the Heartbleed [3] bug. They further illustrate the complexities involved in detecting this bug in its original manifestation. In an early work, Ramalingam et al [71] develop a precise formulation of the problem of determining the likelihood of an analysis fact holding true during execution. This formulation associates a probability with each edge of a control flow graph, which they suggest can be estimated by dynamic techniques. This paper motivated the instantiation of our approach in Presto.

Techniques that take a statistical approach. Such techniques leverage various kinds of program features to statistically determine which alarms are likely bugs. The z-ranking algorithm [42, 41] uses the observation that alarms within physical proximity of each other (e.g., within the same function or file) are correlated in their ground truth and applies a statistical technique called the z-test to rank alarms. More recently, other kinds of program features have been used to statistically classify analysis alarms [39, 11, 34, 77]. Further out, there is a large body of work on using statistical techniques for mining likely specifications and reporting anomalies as bugs (e.g., [60, 52, 43, 72]) and for improving the performance of static analyzers (e.g., [32, 33, 17]). In particular, Banerjee et al [52] propose a new algorithm for automatically inferring explicit information flow specification from program code. In order to infer this, they model information flow paths in a propagation graph constructed from program code, using probabilistic constraints. They solve the system of probabilistic constraints using probabilistic inference on factor graphs.

There has also been extensive research to combine logical and probabilistic reasoning in AI. It starts with Pearl [67, 68], and other examples include Bayesian networks and Markov networks. Koller and Friedman's comprehensive textbook [40] gives a thorough treatment of these techniques. A more recent challenge involves extending these models to capture richer logical formalisms such as Horn clauses and first-order logic. This has resulted in frameworks such as probabilistic relational models [29], Markov logic networks [74, 65], Bayesian logic programs [36], and probabilistic languages such as Blog [57], ProbLog [25], and Infer.NET [58].

There are several methods to perform marginal inference in Bayesian networks. Examples include exact methods, such as variable elimination, the junction tree algorithm [37], and symbolic techniques [28], approximate methods based on belief propagation [44, 59], and those based on sampling, such as Gibbs sampling or MCMC search. Recent advances on the random generation of SAT witnesses [18] also fall in this area. In our work, we use the loopy belief propagation algorithm for discrete approximate inference. Murphy et al [61] describe an empirical study performed to evaluate the loopy belief propagation algorithm as an approximate inference algorithm in a general setting. They found that the algorithm often converges, and when it does, it does converge to a good approximation of the correct marginal as computed by exact inference. They also perform some initial investigation into the cause of oscillations, when the algorithm oscillated without convergence on one probabilistic network. They conclude that simple methods of preventing such oscillations do not allow the algorithm to converge to (approximately) correct marginals.

Chapter 7

Future Directions

While this work demonstrates how probabilistic reasoning can complement program reasoning to improve the effectiveness of automated reasoning tools, it also raises many intriguing research questions. In addition, the generality of this approach opens up many avenues to harness probabilistic reasoning in innovative ways to augment logical reasoning. We discuss below some interesting ways in which this research can be taken forward.

Incorporating extra-analytic features. Our probabilistic model, as defined in this dissertation, only captures the program features recognized by the underlying static program analysis. For example, the datarace analysis executed by Bingo does not recognize or poorly recognizes program features like (a) program control flow paths, (b) access of the fields of the *this* object in a constructor, (c) programmatic constructs like flags and conditionals to avoid racy accesses or (d) intersecting locksets guarding two memory accesses that may potentially race with each other. Failure to capture such program features makes the deductive steps of a static program analysis incomplete, eventually leading the static analysis to report false alarms. While the incomplete deductive steps are modeled by our probabilistic model, the cause of incompleteness, such as the above program features, are not modeled. Therefore, when our probabilistic model recomputes alarm probabilities by conditioning them on evidence of the ground truth obtained by, say, a user, the generalization is sometimes poor and sometimes even erratic. Simply put, when our probabilistic model recomputes alarm probabilities by conditioning them on user feedback, it is possible that it lowers the probability of a true bug, or increases the probability of a false alarm. It is well-known that incorporating such program features into a static program analysis will make the analysis unscalable and practically unusable. Therefore, an interesting research direction will be to explore ways in which extra-analytic program features can be represented in the probabilistic model with a goal to maximize the generalization from user feedback (or, in other words, maximize the generalization from observations). This has the potential to improve alarm ranking.

Improving inference time. Static program analyses that execute at industry scale analyze large and complex programs. As a result, they produce very large derivation graphs, which in turn convert to extremely large Bayesian networks. Inference over such large networks is very time consuming. The size of the network also negatively impacts the convergence of inference algorithms to stable probability distributions. Moreover, in the context of BINGO, marginal inference is performed several times in an iterative manner. The speed of inference is the main reason why BINGO does not qualify as an interactive tool. Therefore, in its current form, it is impractical to incorporate BINGO as part of an integrated software development environment. In this context, substantially improving inference time is a useful direction for research. Improving inference time can be tackled in two ways. One way is to work on the inference engine. In this dissertation, we have used a general off-the-shelf engine to perform discrete approximate inference. We could investigate ways to customize the inference algorithm to the context of program analysis in order to improve inference time. Another way is to optimize the derivation graph or the Bayesian network: to reduce its size until inference time over the resulting network is within acceptable limits. For this, one could investigate principled ways to elide or coalesce parts of the derivation graph that may be determined to be irrelevant to the root cause of a reported alarm. We could also examine if a large monolithic derivation graph could be divided into modular components on which inference could be independently performed, summarized, and later composed.

Learning rule probabilities. In the Bayesian network constructed by our end-to-end system Bingo, we quantify the incompleteness of every deductive step executed by the static analysis, by a fixed number 0.999. That is, we say that if all the antecedent facts of a deductive step hold true, then the consequent, or the conclusion of the deductive step, holds true with a likelihood of 0.999. We make this simplistic initialization to seed the alarm probabilities, and rely on iteratively improving the inferred alarm probabilities by conditioning on evidence. As the deductive rules of a static analysis are not all uniformly incomplete, an interesting research direction is to explore the possibility of learning rule probabilities from training data. If perfectly labelled training data were available, then the rule probability (that captures its incompleteness) will be the fraction of times the rule produced a true conclusion when all its premises held true. Since it is very unlikely that fully labelled data is available, we could use techniques like Expectation Maximization on training data to learn maximum likelihood estimates for rule probabilities.

Incorporating richer inputs from a user or from a dynamic analysis. In our end-to-end system Bingo, we iteratively improve the alarm ranking by seeking feedback from a user. In each iteration, Bingo produces an alarm ranking, seeks feedback from a user for the top-ranked alarm, and recomputes the probabilities of all alarms conditioned on this evidence to produce an improved ranking. In our work, we have taken the greedy approach of seeking user feedback for a single alarm, which is the top-ranked alarm in each iteration. Besides, our implementation seeks only a boolean true/false feedback for each proposed alarm. We could make the interactive loop in Bingo potentially more effective by:

1. Casting the problem of choosing one or more alarms for seeking feedback in each iteration as a problem of finding an optimal strategy in a Markov decision process.

2. Seeking more fine-grained feedback from a user (likely false, likely true, a probability estimate), or a likely explanation in some form (an alarm is false because some other analysis fact is false), in each iteration. In addition to improving alarm ranking, richer feedback may also help mitigate the impact of erroneous feedback.

In our end-to-end system Presto, we seek probability estimates for input analysis facts from a dynamic analysis. We could investigate how to get more precise probability estimates from the dynamic analysis by providing it with more context for each analysis fact. For example, instead of asking the dynamic analysis to estimate how often an analysis fact, say x, holds true, we could ask the dynamic analysis to estimate how often the analysis fact x holds true in the context of a different analysis fact y holding true.

Chapter 8

Concluding Remarks

In this dissertation, we tackle the problem of improving the effectiveness of bug-finding static program analysis tools in practical settings. Specifically, we propose a principled and general approach to augment program reasoning with probabilistic reasoning in order to rank alarms reported by these tools, where the alarms are ranked by their likelihood of being true alarms. This enables tool users to focus their efforts on triaging alarms at the top of the ranking, which are more likely to be true than the alarms lower in the ranking, thereby mitigating the burden of inspecting false alarms.

The approach developed in this dissertation constructs a probabilistic model, which is a Bayesian network, from the deductive steps applied by a static program analysis. The probabilistic model models (a) the dependence of these deductive steps on the analysis facts that they are premised upon, and (b) the dependence of the analysis facts on the deductive steps that derive them. In this manner, the model captures the transitive dependence of analysis facts on other analysis facts, thereby elegantly capturing correlations, or shared dependencies between alarms. Furthermore, the model quantifies the incompleteness of each analysis fact and deductive step. Marginal inference on this probabilistic model associates a posterior probability with each alarm indicating how complete (or, how true) the alarm may be.

We further demonstrate the generality and the empirical effectiveness of this approach by instantiating it in two practical end-to-end systems. The first end-to-end system called Bingo instantiates this approach to rank the alarms reported by two static analyses: the datarace analysis and the taint analysis. Bingo also demonstrates how alarm ranking based on the inferred probabilities iteratively improves when the probabilities are conditioned on evidence gathered by interacting with a human user. The second system called Presto instantiates this approach to rank the alarms reported by an exception flow analysis. Further, Presto demonstrates the feasibility of producing an effective alarm ranking by seeking probability estimates from a dynamic analysis, for incomplete input facts that the analysis is premised upon.

In the last part of the dissertation, we explore the important problem of eliminating directed cycles from derivation graphs that capture the deductive steps applied by a static analysis. We give three algorithms for cycle elimination that are increasingly precise in the number of derivation trees they retain.

In conclusion, the approach of augmenting logical program reasoning with probabilistic reasoning is a very general approach to associate richer information with the conclusions drawn by a static analysis. Moreover, this approach gives us a principled way to tailor static analyses to individual codebases and user needs. As demonstrated in the dissertation, this approach can be harnessed to make static program analyses more effective in practical environments.

Bibliography

[1] Task-based Asynchronous Pattern. https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap.

[2] Clang Static Analyzer. https://clang-analyzer.llvm.org/, 2009.

[3] The Heartbleed Bug. https://heartbleed.com/, 2012.

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

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

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

[7] Thomas Ball. The concept of dynamic analysis. In *Software Engineering ESEC/FSE 1999. ESEC 1999, SIGSOFT FSE 1999. Lecture Notes in Computer Science, vol 1687.*, ESEC/FSE 1999, pages 216–234. Springer, Berlin, Heidelberg, 1999.

[8] Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, POPL 2002, pages 1–3. ACM, 2002.

[9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010.

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

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

[12] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the 2003 ACM Conference on Programming Language Design and Implementation*, PLDI 2003, pages 196–207. ACM, 2003.

[13] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom

class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE 2011, pages 241–250. ACM, 2011.

[14] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA âĂŹ09, page 1âĂŞ12, New York, NY, USA, 2009. Association for Computing Machinery.

[15] Martin Bravenboer and Yannis Smaragdakis. 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, pages 243–262. ACM, 2009.

[16] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, pages 3–11. Springer, 2015.

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

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

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[20] Christoph Csallner and Yannis Smaragdakis. Check'n'crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*, pages 422–431, 2005.

[21] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–37, 2008.

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

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

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

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

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

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

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

[29] Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Ben Taskar. Probabilistic relational models. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, pages 129–174. MIT Press, August 2007.

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

[31] Todd Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th ACM Symposium on Principles of Database Systems*, PODS 2007, pages 31–40. ACM, 2007.

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

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

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

[35] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In Wener Damm and Holger Hermanns, editors, *Computer Aided Verification: 19th International Conference, Proceedings*, CAV 2007, pages 226–239. Springer, 2007.

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

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

[38] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. Combining static and dynamic analyses for vulnerability detection: illustration on heartbleed. In *Haifa Verification Conference*, pages 39–50. Springer, 2015.

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

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

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

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

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

[44] Frank Kschischang, Brendan Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, Feb 2001.

[45] Sulekha Kulkarni. DAFFODIL: Datalog Analysis Framework For Dotnet IL. https://github.com/sulekhark/DAFFODIL.git, 2020.

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

[47] Wei Le and Mary Lou Soffa. Generating analyses for detecting faults in path segments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 320–330, 2011.

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

[49] Vasilica Lepar and Prakash P. Shenoy. A comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer architectures for computing marginals of probability distributions. https://arxiv.org/ftp/arxiv/papers/1301/1301.7394.pdf, 1998.

[50] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. Dynamically validating static memory leak warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 112–122, 2013.

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

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

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

[54] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 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, pages 194–208. ACM, 2016.

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

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

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

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

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

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

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

[62] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166. IEEE, 2016.

[63] Mayur Naik. Chord: A program analysis platform for Java. http://jchord.googlecode.com/, 2006.

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

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

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

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

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

[69] David Hovemeyer William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, December 2004.

[70] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 722–735. ACM, 2018.

[71] Ganesan Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI 96, pages 267–277. ACM, 1996.

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

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

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

[75] Mark S Sherriff, Sarah Smith Heckman, J Michael Lake, and Laurie A Williams. Using groupings of static analysis alerts to identify files likely to contain field failures. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, pages 565–568, 2007.

[76] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool, Synthesis lectures on data management vol. 3 no. 2 edition, 2011.

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

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

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

[80] Dalin Zhang, Dahai Jin, Yunzhan Gong, and Hailong Zhang. Diagnosis-oriented alarm correlations. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 172–179. IEEE, 2013.

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

[82] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pages 365–376. ACM, 2013.