Abstract
Programs often evolve by continuously integrating changes from multiple programmers. The effective adoption of program analysis tools in this continuous integration setting is hindered by the need to only report alarms relevant to a particular program change. We present a probabilistic framework, Drake, to apply program analyses to continuously evolving programs. Drake is applicable to a broad range of analyses that are based on deductive reasoning. The key insight underlying Drake is to compute a graph that concisely and precisely captures differences between the derivations of alarms produced by the given analysis on the program before and after the change. Performing Bayesian inference on the graph thereby enables to rank alarms by likelihood of relevance to the change. We evaluate Drake using Sparrow—a static analyzer that targets buffer-overrun, format-string, and integer-overflow errors—on a suite of ten widely-used C programs each comprising 13k–112k lines of code. Drake enables to discover all true bugs by inspecting only 30 alarms per benchmark on average, compared to 85 (3× more) alarms by the same ranking approach in batch mode, and 118 (4× more) alarms by a differential approach based on syntactic masking of alarms which also misses 4 of the 26 bugs overall.

CCS Concepts • Software and its engineering → Automated static analysis; Software evolution; • Mathematics of computing → Bayesian networks.

Keywords Static analysis, software evolution, continuous integration, alarm relevance, alarm prioritization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI ’19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314616

1 Introduction
The application of program analysis tools such as Astrée [5], SLAM [2], Coverity [4], FindBugs [22], and Infer [7] to large software projects has highlighted research challenges at the intersection of program reasoning theory and software engineering practice. An important aspect of long-lived, multi-developer projects is the practice of continuous integration, where the codebase evolves through multiple versions which are separated by incremental changes. In this context, programmers are typically less worried about the possibility of bugs in existing code—which has been in active use in the field—and in parts of the project which are unrelated to their immediate modifications. They specifically want to know whether the present commit introduces new bugs, regressions, or breaks assumptions made by the rest of the codebase [4, 50, 57]. How do we determine whether a static analysis alarm is relevant for inspection given a small change to a large program?

A common approach is to suppress alarms that have already been reported on previous versions of the program [4, 16, 19]. Unfortunately, such syntactic masking of alarms has a great risk of missing bugs, especially when the commit modifies code in library routines or in commonly used helper methods, since the new code may make assumptions that are not satisfied by the rest of the program [44]. Therefore, even alarms previously reported and marked as false positives may potentially need to be inspected again.

In this paper, we present a probabilistic framework to apply program analyses to continuously evolving programs.
The framework, called Drake, must address four key challenges to be effective. First, it must overcome the limitation of syntactic masking by reasoning about how semantic changes impact alarms. For this purpose, it employs derivations—produced by the given analysis on the program before and after the change. Such derivations are naturally obtained from analyses whose reasoning can be expressed or instrumented via deductive rules. As such, Drake is applicable to a broad range of analyses, including those commonly specified in the logic programming language Datalog [6, 46, 59].

Second, Drake must relate abstract states of the two program versions which do not share a common vocabulary. We build upon previous syntactic program differencing work by setting up a matching function which maps source locations, variable names, and other syntactic entities of the old version of the program to the corresponding entities of the new version. The matching function allows not only to relate alarms but also the derivations that produce them.

Third, Drake must efficiently and precisely compute the relevance of each alarm to the program change. For this purpose, it constructs a differential derivation graph that captures differences between the derivations of alarms produced by the given analysis on the program before and after the change. For a fixed analysis, this graph construction takes effectively linear time, and it captures all derivations of each alarm in the old and new program versions.

Finally, Drake must be able to rank the alarms based on likelihood of relevance to the program change. For this purpose, we leverage recent work on probabilistic alarm ranking [53] by performing Bayesian inference on the graph. This approach also enables further to improve the ranking by taking advantage of any alarm labels provided by the programmer offline in the old version and online in the new version of the program.

We have implemented Drake and demonstrate how to apply it to two analyses in Sparrow [49], a sophisticated static analyzer for C programs: an interval analysis for buffer-overflow errors, and a taint analysis for format-string and integer-overflow errors. We evaluate the resulting analyses on a suite of ten widely-used C programs each comprising 13k–112k lines of code, using recent versions of these programs involving fixes of bugs found by these analyses. We compare Drake’s performance to two state-of-the-art baseline approaches: probabilistic batch-mode alarm ranking [53] and syntactic alarm masking [50]. To discover all the true bugs, the Drake user has to inspect only 30 alarms on average per benchmark, compared to 85 (3× more) alarms and 118 (4× more) alarms by each of these baselines, respectively. Moreover, syntactic alarm masking suppresses 4 of the 26 bugs overall. Finally, probabilistic inference is very unintrusive, and only requires an average of 25 seconds to re-rank alarms after each round of user feedback.

### Contributions

In summary, we make the following contributions in this paper:

1. We propose a new probabilistic framework, Drake, to apply static analyses to continuously evolving programs. Drake is applicable to a broad range of analyses that are based on deductive reasoning.
2. We present a new technique to relate static analysis alarms between the old and new versions of a program. It ranks the alarms based on likelihood of relevance to the difference between the two versions.
3. We evaluate Drake using different static analyses on widely-used C programs and demonstrate significant improvements in false positive rates and missed bugs.

### 2 Motivating Example

We explain our approach using the C program shown in Figure 1. It is an excerpt from the audio file processing utility shntool, and highlights changes made to the code between versions 3.0.4 and 3.0.5, which we will call $P_{old}$ and $P_{new}$ respectively. Lines preceded by a “+” indicate code which has been added, and lines preceded by a “−” indicate code which has been removed from the new version. The integer overflow analysis in Sparrow reports two alarms in each version of this code snippet, which we describe next.

The first alarm, reported at line 30, concerns the command line option “t”. This program feature trims periods of silence from the ends of an audio file. The program reads unsanitized data into the field info->header_size at line 25, and allocates a buffer of proportional size at line 30. Sparrow observes this data flow, concludes that the multiplication could overflow, and subsequently raises an alarm at the allocation site. However, this data has been sanitized at line 29, so that the expression header_size * sizeof(char) cannot overflow. This is therefore a false alarm in both $P_{old}$ and $P_{new}$. We will refer to this alarm as Alarm(30).

The second alarm is reported at line 45, and is triggered by the command line option “c”. This program feature compares the contents of two audio files. The first version has source-sink flows from the untrusted fields info1->data_size and info2->data_size, but this is a false alarm since the value of bytes cannot be larger than CMP_SIZE. On the other hand, the new version of the program includes an option to offset the contents of one file by shift_secs seconds. This value is used without sanitization to compute cmp_size, leading to a possible integer overflow at line 42, which would then result in a buffer of unexpected size being allocated at line 45. Thus, while Sparrow raises an alarm at the same allocation site for both versions of the program, which we will call Alarm(45), this is a false alarm in $P_{old}$ but a real bug in $P_{new}$.

We now restate the central question of this paper: How do we alert the user to the possibility of a bug at line 45, while not forcing them to inspect all the alarms of the “batch mode” analysis, including that at line 30?
Figure 1. An example of a code change between two versions of the audio processing utility shntool. Lines 1 and 41 have been removed, while lines 3, 42, 43, and 54 have been added. In the new version, the use of the unsanitized value shift_secs can result in an integer overflow at line 42, and consequently result in a buffer of unexpected size being allocated at line 45.

The input tuples indicate elementary facts about the program which the analyzer determines from the program text. For example, the tuple DUEdge(7, 9) indicates that there is a one-step data flow from line 7 to line 9 of the program. The inference rules, which we express here as Datalog programs, provide a mechanism to derive new conclusions about the program being analyzed. For example, the rule r2, DUPath(c1, c3) = DUPath(c1, c2), DUEdge(c2, c3), indicates that for each triple (c1, c2, c3) of program points, whenever there is a multi-step data flow from c1 to c2 and an immediate data flow from c2 to c3, there may be a multi-step data flow from c1 to c3. Starting from the input tuples, we repeatedly apply these inference rules to reach new conclusions, until we reach a fixpoint. This process may be visualized as discovering the nodes of a derivation graph such as that shown in Figure 4.

We use derivation graphs to determine alarm relevance. As we have just shown, such derivation graphs can be naturally described by inference rules. These inference rules are straightforward to obtain if the analysis is written in a declarative language such as Datalog. If the analysis is written in a general-purpose language, we define a set of inference rules that approximate the reasoning processes.
of the original analyzer. The degree of approximation does not affect the accuracy of the analysis but only affects the accuracy of subsequent probabilistic reasoning. Furthermore, in practice, it requires only a small amount of effort to implement by instrumenting the original analyzer. We explain this instrumentation for a general class of analyses in Section 4.2.

### 2.2 Classifying Derivations by Alarm Transfer

Traditional approaches such as syntactic alarm masking will deprioritize both Alarm(30) and Alarm(45) as they occur in both versions of the program. Concretely then, our problem is to provide a mechanism by which to continue to deprioritize Alarm(30), but highlight Alarm(45) as needing reinspection.

**Translating clauses.** For each grounded clause $g$ in the derivation from the new program $P_{new}$, we can ask whether $g$ also occurs in the old program $P_{old}$. For example, the clauses in Figure 4(a) commonly exist in both of the versions, but the clauses in Figure 4(c) are only present in $P_{new}$. Such questions presuppose the existence of some correspondence between program points, variables, functions, and other syntactic entities of $P_{old}$ and the corresponding entities of $P_{new}$. In Section 4.3, we will construct a matching function $\delta$ to perform this translation, but for the purpose of this example, it can be visualized as simply being a translation between line numbers, such as that obtained using $\text{diff}$.

**Translating derivation trees.** The graph of Figure 4 can be viewed as encoding a set of derivation trees for each alarm. A derivation tree is an inductive structure which culminates in the production of a tuple $t$. It is either: (a) an input tuple, or (b) a grounded clause $t_1 \land t_2 \land \cdots \land t_k \implies t$ together with a derivation tree $t_i$ for each antecedent tuple $t_i$.

Let us focus on two specific derivation trees from this graph: first, the sequence $r_{30}$ in Figure 4(a): 

$$\text{DUPath}(7, 9) \rightarrow \text{DUPath}(7, 18) \rightarrow \cdots \rightarrow \text{Alarm}(30),$$

and second, the sequence $r_{45}$ in Figure 4(c): 

$$\text{DUPath}(54, 42) \rightarrow \text{DUPath}(54, 43) \rightarrow \cdots \rightarrow \text{Alarm}(45),$$

and where each sequence is supplemented with appropriate input tuples. Observe that each clause of the first tree, $r_{30}$, is common to both $P_{old}$ and $P_{new}$. More generally, every derivation tree of Alarm(30) from $P_{new}$ is already present in $P_{old}$. As a result, Alarm(30) is unlikely to represent a real bug. On the other hand, the second tree, $r_{45}$, exclusively occurs in the new version of the program. Therefore, since there are more reasons to suspect the presence of a bug at Alarm(45) in $P_{new}$ than in $P_{old}$, we conclude that it is necessary to reinspect this alarm.

The first step to identifying relevant alarms is therefore to determine which alarms have new derivation trees. As we show in Figure 5, where the new $t_2 \rightarrow t_3$ derivation for $t_3$...
Deleting clauses common to both versions—

\[ \text{Figure 5.} \]

for some \( \tau \), in the inductive case, the new analysis run iff it does not appear in the old program.

The differential derivation graph. Notice that a derivation tree \( \tau \) is either an input tuple \( t \) or a grounded clause \( t_1 \land t_2 \land \cdots \land t_k \implies \tau \) applied to a set of smaller derivation trees \( t_1, t_2, \ldots, t_k \). If \( \tau \) is an input tuple, then it is exclusive to the new analysis run if it does not appear in the old program. In the inductive case, \( \tau \) is exclusive to the new version iff, for some \( i \), the sub-derivation \( t_i \) is in turn exclusive to \( P_{\text{new}} \).

Transitive extension to \( t_4 \), this question inherently involves non-local reasoning. Other approaches based on enumerating derivation trees by exhaustive unrolling of the fixpoint graph will fail in the presence of loops, i.e., when the number of derivation trees is infinite. For a fixed analysis, we will now describe a technique to answer this question in time linear in the size of the new graph.

For example, consider the tuple \( \text{DUPath}(7, 18) \) from Figure 4(a), which results from an application of the rule \( r_2 \) to the tuples \( \text{DUPath}(7, 9) \) and \( \text{DUEdge}(9, 18) \):

\[ g = \text{DUPath}(7, 9) \land \text{DUEdge}(9, 18) \implies r_2 \quad \text{DUPath}(7, 18). \] (1)

Observe that \( g \) is the only way to derive \( \text{DUPath}(7, 18) \), and that both its hypotheses \( \text{DUPath}(7, 9) \) and \( \text{DUEdge}(9, 18) \) are common to \( P_{\text{old}} \) and \( P_{\text{new}} \). As a result, \( P_{\text{new}} \) does not contain any new derivations of \( \text{DUPath}(7, 18) \).

On the other hand, consider the tuple \( \text{DUPath}(7, 42) \) in Figure 4(c), which results from the following application of \( r_2 \):

\[ g' = \text{DUPath}(7, 39) \land \text{DUEdge}(39, 42) \implies r_2 \quad \text{DUPath}(7, 42). \] (2)

and notice that its second hypothesis \( \text{DUEdge}(39, 42) \) is exclusive to \( P_{\text{new}} \). As a result, \( \text{DUPath}(7, 42) \), and all its downstream consequences including \( \text{DUPath}(7, 43), \text{DUPath}(7, 45) \), and \( \text{Alarm}(45) \) possess derivation trees which are exclusive to \( P_{\text{new}} \).

Our key insight is that we can perform this classification of derivation trees by splitting each tuple \( t \) into two variants, \( t_a \) and \( t_b \). We set this up so that the derivations of \( t_a \) correspond exactly to the trees which are common to both versions, and the derivations of \( t_b \) correspond exactly to the trees which are exclusive to \( P_{\text{new}} \). For example, the clause \( g \) splits into four copies, \( g_{aa}, g_{a\beta}, g_{\beta a}, g_{\beta\beta} \), for each combination of
Figure 6. Differentiating the clause \( g \) from Equation 1.

antecedents:

\[
g_{\alpha \alpha} = DUPath_{\alpha}(7, 9) \land DUEdge_{\alpha}(9, 18) \\
\implies r_2 DUPath_{\alpha}(7, 18), \quad (3)
\]

\[
g_{\alpha \beta} = DUPath_{\alpha}(7, 9) \land DUEdge_{\beta}(9, 18) \\
\implies r_2 DUPath_{\beta}(7, 18), \quad (4)
\]

\[
g_{\beta \alpha} = DUPath_{\beta}(7, 9) \land DUEdge_{\alpha}(9, 18) \\
\implies r_2 DUPath_{\alpha}(7, 18), \quad (5)
\]

\[
g_{\beta \beta} = DUPath_{\beta}(7, 9) \land DUEdge_{\beta}(9, 18) \\
\implies r_2 DUPath_{\beta}(7, 18). \quad (6)
\]

Observe that the only way to derive \( DUPath_{\alpha}(7, 18) \) is by applying a clause to a set of tuples all of which are themselves of the \( \alpha \)-variety. The use of even a single \( \beta \)-variant hypothesis always results in the production of \( DUPath_{\beta}(7, 18) \). We visualize this process in Figure 6. By similarly splitting each clause \( g \) of the analysis fixpoint, we produce the clauses of the differential derivation graph \( GC_\alpha \).

At the base case, let the set of merged input tuples \( I_\alpha \) be the \( \alpha \)-variants of input tuples which occur in common, and the \( \beta \)-variants of all input tuples which only occur in \( P_\text{new} \). Observe then that, since there are no new dataflows from lines 7 to 9, only \( DUPath_{\alpha}(7, 9) \) is derivable but \( DUPath_{\beta}(7, 9) \) is not. Furthermore, since \( DUEdge_{\beta}(9, 18) \) is common to both program versions, we only include its \( \alpha \)-variant, \( DUEdge_{\alpha}(9, 18) \) in \( I_\alpha \), and exclude \( DUEdge_{\beta}(9, 18) \). As a result, both hypotheses of \( g_{\alpha \alpha} \) are derivable, so that \( DUPath_{\alpha}(7, 18) \) is also derivable, but at least one hypothesis of its sibling clauses, \( g_{\alpha \beta}, g_{\beta \alpha}, \) and \( g_{\beta \beta} \), are underviable, so that \( DUPath_{\beta}(7, 18) \) also fails to be derivable. By repeating this process, \( GC_\alpha \) permits us to conclude the derivability of \( Alarm_{\alpha}(30) \) and the non-derivability of \( Alarm_{\beta}(30) \).

In contrast, the hypothesis \( DUEdge(39, 42) \) of \( g' \) is only present in \( P_\text{new} \), so that we include \( DUEdge_{\alpha}(39, 42) \) in \( I_\alpha \), but exclude its \( \alpha \)-variant. As a result, \( g_{\alpha \beta} = DUPath_{\alpha}(7, 39) \land DUEdge_{\beta}(39, 42) \implies r_2 DUPath_{\beta}(7, 42) \) successfully fires, but all of its siblings—\( g_{\alpha \alpha}, g_{\beta \alpha}, \) and \( g_{\beta \beta} \)—are inactive. The differential derivation graph, \( GC_\alpha \), thus enables the successful derivation of \( DUPath_{\beta}(7, 42) \), and of all its consequences, \( DUPath_{\beta}(7, 43), DUPath_{\beta}(7, 45) \), and \( Alarm_{\beta}(45) \).

2.3 A Probabilistic Model of Alarm Relevance

We build our system on the idea of highlighting alarms \( Alarm(c) \) whose \( \beta \)-variants, \( Alarm_{\beta}(c) \), are derivable in the differential derivation graph. By leveraging recent work on probabilistic alarm ranking [53], we can also transfer feedback across program versions and highlight alarms which are both relevant and likely to be real bugs. The idea is that since alarms share root causes and intermediate tuples, labelling one alarm as true or false should change our confidence in closely related alarms.

**Differential derivation graphs, probabilistically.** The inference rules of the analysis are frequently designed to be sound, but deliberately incomplete. Let us say that a rule *misfires* if it takes a set of true hypotheses, and produces an output tuple which is actually false. In practice, in large real-world programs, rules misfire in statistically regular ways. We therefore associate each rule \( r \) with the probability \( p_r \) of its producing valid conclusions when provided valid hypotheses.

Consider the rule \( r_3 \) and its instantiation as the grounded clause in Figure 6, \( g_{\alpha \beta} = r_3(t_1, t_2) \), with \( t_1 = DUPath_{\alpha}(7, 9) \) and \( t_2 = DUEdge_{\beta}(9, 18) \) as its antecedent tuples, and with \( t_3 = DUPath_{\beta}(7, 18) \) as its conclusion. We define:

\[
Pr(g_{\alpha \beta} | t_1 \land t_2) = p_{r_3}, \quad (7)
\]

\[
Pr(g_{\alpha \beta} | \neg t_1 \lor \neg t_2) = 0, \quad (8)
\]

so that \( g_{\alpha \beta} \) successfully fires only if \( t_1 \) and \( t_2 \) are both true, and even in that case, only with probability \( p_{r_3} \).

The conclusion \( t_3 \) is true iff any one of its deriving clauses successfully fires:

\[
Pr(t_3 | g_{\alpha \beta} \lor g_{\beta \alpha} \lor g_{\beta \beta}) = 1, \quad (9)
\]

\[
Pr(t_3 | \neg(g_{\alpha \beta} \lor g_{\beta \alpha} \lor g_{\beta \beta}) = 0. \quad (10)
\]

Finally, we assign high probabilities (\( \approx 1 \)) to input tuples \( t \in I_\alpha \) (e.g., \( DUEdge_{\alpha}(7, 9) \)) and low probabilities (\( \approx 0 \)) to input tuples \( t \notin I_\alpha \) (e.g., \( DUEdge_{\beta}(7, 9) \)). As a result, the \( \beta \)-variant of each alarm, \( Alarm_{\beta}(c) \), has a large prior probability, \( Pr(Alarm_{\beta}(c)) \), in exactly the cases where it is possible to add the missing new derivation trees in \( P_\text{new} \) and is thus likely to be relevant to the code change. In particular, \( Pr(Alarm_{\beta}(45)) \gg Pr(Alarm_{\beta}(30)) \), as we originally desired.

**Interaction Model.** Drake presents the user with a list of alarms, sorted according to \( Pr(Alarm(c) \mid e) \), i.e., the probability that \( Alarm(c) \) is both relevant and a true bug, conditioned on the current feedback set \( e \). After each round of user feedback, we update \( e \) to include the user label for the last triaged alarm, and rerank the remaining alarms according to \( Pr(Alarm(c) \mid e) \).

Furthermore, \( e \) can also be initialized by applying any feedback that the user has provided to the old program, *precommit*, say to \( Alarm(45) \), to the old versions of the corresponding tuples in \( GC_\alpha \), i.e., to \( Alarm_{\alpha}(45) \). We note that this

1There are various ways to obtain these rule probabilities, but as pointed out by [53], heuristic judgments, such as uniformly assigning \( p_r = 0.99 \), work well in practice.
We formally describe the Drake Datalog program. Outputs all immediate dataflows, analysis (connect them. We obtain GC set of alarms, and the set of input facts, \( A \_\text{analysis result} \). Declarative program analysis.

### 3.1 Preliminaries

**Declarative program analysis.** Drake assumes that the analysis result \( \mathcal{A}(P) \) is a tuple, \( \mathbf{R} = (I, C, A, GC) \), where \( I \) is the set of input facts, \( C \) is the set of output tuples, \( A \) is the set of alarms, and \( GC \) is the set of grounded clauses which connect them. We obtain \( I \) by instrumenting the original analysis \( (A, I) = \mathcal{A}_{\text{orig}}(P) \). For example, in our experiments, Sparrow outputs all immediate dataflows, DUEdge\((c_1, c_2)\) and potential source and sink locations, Src\((c)\) and Dst\((c)\). We obtain \( C \) and \( GC \) by approximating the analysis with a Datalog program.

A Datalog program [1]—such as that in Figure 3—consumes a set of input relations and produces a set of output relations. Each relation is a set of tuples, and the computation of the output relations is specified using a set of rules. A rule \( r \) is an expression of the form \( R_h(\mathbf{v}_h) \leftarrow R_1(\mathbf{v}_1), R_2(\mathbf{v}_2), \ldots, R_k(\mathbf{v}_k) \), where \( R_1, R_2, \ldots, R_k \) are relations, \( R_h \) is an output relation, \( \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \) and \( \mathbf{v}_h \) are vectors of variables of appropriate arity. The rule \( r \) encodes the following universally quantified logical formula: "For all values of \( \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \) and \( \mathbf{v}_h \), if \( R_1(\mathbf{v}_1) \land R_2(\mathbf{v}_2) \land \cdots \land R_k(\mathbf{v}_k) \), then \( R_h(\mathbf{v}_h) \)."

To evaluate the Datalog program, we initialize the set of conclusions \( C := I \) and the set of grounded clauses \( GC := \emptyset \), and repeatedly instantiate each rule to add tuples to \( C \) and grounded clauses to \( GC \): i.e., whenever \( R_1(c_1), R_2(c_2), \ldots, R_k(c_k) \in C \), we update \( C := C \cup \{ R_h(c_h) \} \) and \( GC := GC \cup \{ R_1(c_1) \land R_2(c_2) \land \cdots \land R_k(c_k) \implies R_h(c_h) \} \).

For each grounded clause \( g \) of the form \( H_g \implies c_g \), we refer to \( H_g \) as the set of antecedents of \( g \) and \( c_g \) as its conclusion. We repeatedly add tuples to \( C \) and grounded clauses to \( GC \) until a fixpoint is reached.

**Bayesian alarm ranking.** The main observation behind Bayesian alarm ranking [33] is that alarms are correlated in their ground truth: labelling one alarm as true or false should change our confidence in the tuples involved in its production, and transitively, affect our confidence in a large number of other related alarms. Concretely, these correlations are encoded by converting the set of grounded clauses \( GC \) into a Bayesian network: we will now describe this process.

Let \( G \) be the derivation graph formed by all tuples \( t \in C \) and grounded clauses \( g \in GC \). Figure 4 is an example. Consider a grounded clause \( g \in GC \) of the form \( t_1 \land t_2 \land \cdots \land t_k \implies t_k \). Observe that \( g \) requires all its antecedents to be true to be able to successfully derive its output tuple. In particular, if any of the antecedents fails, then the clause is definitely inoperative. Let us assume a function \( p \) which maps each rule \( r \) to the probability of its successful firing, \( p_r \). Then, we associate \( g \) with the following conditional probability distribution (CPD) using an assignment \( \mathcal{P} \):

\[
\mathcal{P}(g \mid t_1 \land t_2 \land \cdots \land t_k) = p_r, \quad \text{and} \quad \mathcal{P}(g \mid \neg(t_1 \land t_2 \land \cdots \land t_k)) = 0.
\]

The conditional probabilities of an event and its complement sum to one, so that \( \Pr(\neg g \mid t_1 \land t_2 \land \cdots \land t_k) = 1 - p_r \), and \( \Pr(\neg g \mid \neg(t_1 \land t_2 \land \cdots \land t_k)) = 1 \).

On the other hand, consider some tuple \( t \) which is produced by the clauses \( g_1, g_2, \ldots, g_l \). If there exists some clause \( g_i \) which is derivable, then \( t \) is itself derivable. If none of the clauses is derivable, then neither is \( t \). We therefore associate \( t \) with the CPD for a deterministic disjunction:

\[
\mathcal{P}(t \mid g_1 \lor g_2 \lor \cdots \lor g_l) = 1, \quad \text{and} \quad \mathcal{P}(t \mid \neg(g_1 \lor g_2 \lor \cdots \lor g_l)) = 0.
\]
Let us also assume a function $p_m$ which maps input tuples $t$ to their prior probabilities. In the simplest case, input tuples are known with certainty, so that $p_m(t) = 1$. In Section 3.2, we will see that the choice of $p_m$ allows us to uniformly generalize both relevance-based and traditional batch-mode ranking. We define the CPD of each input tuple $t$ as:

$$P(t) = p_m(t).$$  \hfill (16)

By definition, a Bayesian network is a pair $(G, P)$, where $G$ is an acyclic graph and $P$ is an assignment of CPDs to each node \cite{kaissar-et-al-2013}. We have already defined the CPDs in Equations 12–16; the challenge is that the derivation graph $G$ may have cycles. Raghothaman et al. \cite{raghothaman-et-al-2012} present an algorithm to extract an acyclic subgraph $G_c \subseteq G$ which still preserves derivability of all tuples. Using this, we may define the final Bayesian network, $Bnet(R) = (G_c, P)$.

### 3.2 The Constraint Merging Process

As motivated in Section 2.2, we combine the constraints from the old and new analysis runs into a single differential derivation graph $R_{\delta}$. Every derivation tree $\tau$ of a tuple from $R_{new}$ is either common to both $R_{\delta}$ and $R_{new}$, or is exclusive to the new analysis run.

Recall that a derivation tree is inductively defined as either: (a) an individual input tuple, or (b) a grounded clause $t_1 \land t_2 \land \cdots \land t_k \implies r$ of the form $t_\alpha \land t_\beta \land \cdots \land t_\kappa \implies r$ which only occur in $R_{new}$ of new derivation trees for at least one child $t_i$.

The idea behind the construction of $R_{\delta}$ is therefore to split each tuple $t$ into two variants, $t_\alpha$ and $t_\beta$, where $t_\alpha$ precisely captures the common derivation trees and $t_\beta$ captures the derivation trees which only occur in $R_{new}$. We formally describe its construction in Algorithm 2. Theorem 3.1 is a straightforward consequence.

**Theorem 3.1 (Separation).** Let the combined analysis results from $P_{old}$ and $P_{new}$ be $R_{\Lambda} = Merge(R_{\delta}, R_{new})$. Then, for each tuple $t$,

1. $t_\alpha$ is derivable from $R_{\Lambda}$ if and only if $t$ has a derivation tree which is common to both $R_{\delta}$ and $R_{new}$.
2. $t_\beta$ is derivable from $R_{\Lambda}$ if and only if $t$ has a derivation tree which is absent from $R_{\delta}$ but present in $R_{new}$.

**Proof:** In each case, by induction on the tree which is given to exist. All base cases are all immediate. We will now explain the inductive cases.

Of part 1, in the $\implies$ direction. Let $t_\alpha$ be the result of a clause $t_1 \land t_2 \land \cdots \land t_k \implies r$, By construction, it is the case that each $t_i$ is of the form $t_\alpha$, and by IH, it must already have a derivation tree $\tau_i$ which is common to both analysis results. It follows that $t_\alpha$ also has a derivation tree $r(\tau_1, \tau_2, \ldots, \tau_k)$ in common to both results.

In the $\impliedby$ direction. Similarly, let $t_\beta$ be the result of a clause $t_1 \land t_2 \land \cdots \land t_k \implies r$, By construction, it is the case that each $t_i$ is of the form $t_\beta$, and by IH, it must also have a derivation tree $\tau_i$ which is common to both analysis results.

Notice that the time and space complexity of Algorithm 2 is bounded by the size of the analysis rather than the program being analyzed. If $k_{\text{max}}$ is the size of the largest rule body, then the algorithm runs in $O(2^{k_{\text{max}}})$ time and produces $R_{\Lambda}$ which is also of size $O(2^{k_{\text{max}}}).$ Given a tuple $t \in C_{\text{new}},$ the existence of a derivation tree exclusive to $R_{new}$ can be determined using Theorem 3.1 in time $O(|R_{\Lambda}|).$
computations can be executed in time which is effectively linear in the size of the program.

**Distinguishing abstract derivations.** One detail is that since the output tuples indicate program behaviors in the abstract domain, it may be possible for $P_{\text{new}}$ to have a new concrete behavior, while the analysis continues to produce the same set of tuples. This could conceivably affect ranking performance by suppressing real bugs in $R_\Delta$. Therefore, instead of using $I_\Delta$ as the set of input tuples in $\text{BNET}(R_\Delta)$, we use the set of all input tuples $t \in \{t_{\alpha}, t_{\beta} \mid t \in I_{\text{new}}\}$, with prior probability: if $t \in I_{\text{new}} \setminus I_\Delta$, then $p_m(t_\alpha) = 1 - p_m(t_\beta) = 1.0$, and otherwise, if $t \in I_{\text{new}} \cap I_\Delta$, then $p_m(t_\beta) = 1 - p_m(t_\alpha) = \epsilon$. Here, $\epsilon$ is our belief that the same abstract state has new concrete behaviors. The choice of $\epsilon$ also allows us to interpolate between purely change-based ($\epsilon = 0$) and purely batch-mode ranking ($\epsilon = 1$).

### 3.3 Bootstrapping by Feedback Transfer

It is often the case that the developer has already inspected some subset of the analysis results on the program from before the code change. By applying this old feedback $e_{\text{old}}$ to the new program, as we will now explain, the differential derivation graph also allows us to further improve the alarm rankings beyond just the initial estimates of relevance.

**Conservative mode.** Consider some negatively labelled alarm $\neg a \in e_{\text{old}}$. The programmer has therefore indicated that all of its derivation trees in $R_{\text{old}}$ are false. If $a' = \delta(a)$, since the derivation trees of $a'_n$ in $R_\Delta$ correspond to a subset of the derivation trees of $a$ in $R_{\text{old}}$, we can additionally de-prioritize these derivation trees by initializing:

$$e := \{\neg a_{\alpha} \mid \forall \text{ negative labels } \neg a \in \delta(e_{\text{old}})\}. \quad (17)$$

**Strong mode.** In many cases, programmers have a lot of trust in $P_{\text{old}}$ since it has been tested in the field. We can then make the strong assumption that $P_{\text{old}}$ is bug-free, and extend inter-version feedback transfer, by initializing:

$$e := \{\neg a_{\alpha} \mid \forall a \in A_\Delta\}. \quad (18)$$

Our experiments in Section 5 are primarily conducted with this setting.

**Aggressive mode.** Finally, if the programmer is willing to accept a greater risk of missed bugs, then we can be more aggressive in transferring inter-version feedback:

$$e := \{\neg a_{\alpha}, \neg a_{\beta} \mid \forall a \in A_\Delta\}. \quad (19)$$

In this case, we not only assume that all common derivations of the alarms are false, but also additionally assume that the new alarms are false. It may be thought of as a combination of syntactic alarm masking and Bayesian alarm prioritization. We also performed experiments with this setting and, as expected, observed that it misses 4 real bugs (15%), but additionally reduces the average number of alarms to be inspected before finding all true bugs from 30 to 22.

### 4 Implementation

In this section, we discuss key implementation aspects of Drake, in particular: (a) extracting derivation trees from program analyzers that are not necessarily written in a declarative language, and (b) comparing two versions of a program. In Section 4.2, we explain how we extract derivation trees from complex, black-box static analyses, while Section 4.3 describes the syntactic matching function $\delta$ for a pair of program versions.

#### 4.1 Setting

We assume that the analysis is implemented on top of a sparse analysis framework [48] which is a general method for achieving sound and scalable global static analyzers. The framework is based on abstract interpretation [14] and supports relational as well as non-relational semantic properties for various programming languages.

**Program.** A program is represented as a control flow graph $(C, \rightarrow, c_0)$ where $C$ denotes the set of program points, $(\rightarrow) \subseteq C \times C$ denotes the control flow relation, and $c_0$ is the entry node of the program. Each program point is associated with a command.

**Program analysis.** We target a class of analyses whose abstract domain maps program points to abstract states:

$$D = C \rightarrow S.$$  

An abstract state maps abstract locations to abstract values:

$$S = L \rightarrow V.$$  

The analysis produces alarms for each potentially erroneous program points.

The data dependency relation $(\rightsquigarrow) \subseteq C \times L \times C$ is defined as follows:

$$c_0 \rightsquigarrow l \iff \exists [c_0, c_1, \ldots, c_n] \in \text{Paths}, \exists l \in L. \quad l \in D(c_0) \land U(c_n) \land \forall i \in (0, n). l \notin D(c_i)$$

where $D(c) \subseteq L$ and $U(c) \subseteq L$ denote the def and use sets of abstract locations at program point $c$. A data dependency $c_0 \rightsquigarrow l$ represents that abstract location $l$ is defined at program point $c_0$ and used at $c_n$ through path $[c_0, c_1, \ldots, c_n]$, and no intermediate program points on the path re-define $l$.

#### 4.2 Extracting derivation trees from complex, non-declarative program analyses

To extract the Bayesian network, the analysis additionally computes derivation trees for each alarm. In general, instrumenting a program analyzer to do bookkeeping at each reasoning step would impose a high engineering burden. We instead abstract the reasoning steps using dataflow relations that can be extracted in a straightforward way in static analyses based on the sparse analysis framework [48], including many practical systems [42, 58, 61].
Figure 3 shows the relations and deduction rules to describe the reasoning steps of the analysis. Data flow relation \( \text{DUEdge} \subseteq \mathbb{C} \times \mathbb{C} \) which is a variant of data dependency [48] is defined as follows:

\[
\text{DUEdge}(c_0, c_n) = \exists l \in \mathbb{L}. c_0 \xrightarrow{l} c_n.
\]

A data flow relation \( \text{DUEdge}(c_0, c_n) \) represents that an abstract location is defined at program point \( c_0 \) and used at \( c_n \). Relation \( \text{DUPath}(c_1, c_n) \) represents transitive data flow relation from point \( c_1 \) to \( c_n \). Relation \( \text{Alarm}(c_1, c_n) \) describes an erroneous dataflow from point \( c_1 \) to \( c_n \) where \( c_1 \) and \( c_n \) are the potential origin and crash point of the error, respectively. For a conventional source-sink property (i.e., taint analysis), program points \( c_1 \) and \( c_n \) correspond to the source and sink points for the target class of errors. For other properties such as buffer-overrun that do not fit the source-sink problem formulation, the origin \( c_1 \) is set to the entry point \( c_0 \) of the program and \( c_n \) is set to the alarm point.

### 4.3 Syntactic matching function

To relate program points of the old version \( P_1 \) and the new version \( P_2 \) of the program, we compute function \( \delta \in \mathbb{C}_{P_1} \rightarrow (\mathbb{C}_{P_1} \cup \mathbb{C}_{P_2}) \):

\[
\delta(c_1) = \begin{cases} 
   c_2 & \text{if } c_1 \text{ corresponds to a unique point } c_2 \in \mathbb{C}_{P_2} \\
   c_1 & \text{otherwise}
\end{cases}
\]

where \( \mathbb{C}_{P_1} \) and \( \mathbb{C}_{P_2} \) denote the sets of program points in \( P_1 \) and \( P_2 \), respectively. The function \( \delta \) translates program point \( c_1 \) in the old version to the corresponding program point \( c_2 \) in the new version. If no corresponding program point exists, or multiple possibilities exist, then \( c_1 \) is not translated. In our implementation, we check the correspondence between two program points \( c_1 \) and \( c_2 \) through the following steps:

1. Check whether \( c_1 \) and \( c_2 \) are from the matched file. Our implementation matches the old file with the new file if their names match. This assumption can be relaxed if renaming history is available in a version control system.
2. Check whether \( c_1 \) and \( c_2 \) are from the matched lines. Our implementation matches the old line with the new line using the GNU `diff` utility.
3. Check whether \( c_1 \) and \( c_2 \) have the same program commands. In practice, one source code line can be translated into multiple commands in the intermediate representation of program analyzer.

It is conceivable that our current syntactic matching function, based on `diff`, may perform sub-optimally with tricky semantics-preserving code changes such as statement reorderings. However, we have not observed such complicated changes much in mature software projects. Moreover, we anticipate Drake being used at the level of individual commits or pull-requests that typically change only a few lines of code. In such cases, strong feedback transfer would leave just

### 5 Experimental Evaluation

Our evaluation aims to answer the following questions:

**Q1.** How effective is Drake for continuous and interactive reasoning?

**Q2.** How do different parameter settings of Drake affect the quality of ranking?

**Q3.** Does Drake scale to large programs?

#### 5.1 Experimental Setup

All experiments were conducted on Linux machines with 17 processors running at 3.4 GHz and with 16 GB memory. We performed Bayesian inference using libDAI [45].

**Instance analyses.** We have implemented our system with Sparrow, a static analysis framework for C programs [49]. Sparrow is designed to be soundy [40] and its analysis is flow-, field-sensitive and partially context-sensitive. It basically computes both numeric and pointer values using the interval domain and allocation-site-based heap abstraction. Sparrow has two analysis engines: an interval analysis for buffer-overrun errors, and a taint analysis for format-string and integer-overflow errors. The taint analysis checks whether unchecked user inputs and overflowed integers are used as arguments of printf-like functions and malloc-like functions, respectively. Since each engine is based on different abstract semantics, we run Drake separately on the analysis results of each engine.

We instrumented Sparrow to generate the elementary dataflow relations (DUEdge, Src, and Dst) in Section 4 and used an off-the-shelf Datalog solver Soufflé [25] to compute derivation trees. The dataflow relations are straightforwardly extracted from the sparse analysis framework [48] on which Sparrow is based. Our instrumentation comprises 0.5K lines while the original Sparrow tool comprises 15K lines of OCaml code.

**Benchmarks.** We evaluated Drake on the suite of 10 benchmarks shown in Table 1. The benchmarks include those from previous work applying Sparrow [21] as well as GNU open source packages with recent bug-fix commits. We excluded benchmarks if their old versions were not available. All ground truth was obtained from the corresponding bug reports. Of the 10 benchmarks, 8 bugs were fixed by developers and 4 bugs were also assigned CVE reports. Since commit-level source code changes typically introduce modest semantic differences, we ran our differential reasoning process on two consecutive minor versions of the programs before and after the bugs were introduced.

**Baselines.** We compare Drake to two baseline techniques: Bingo [53] and SynMask. Bingo is an interactive alarm ranking system for batch-mode analysis. It ranks the alarms using
Table 1. Benchmark characteristics. Old and New denote program versions before and after introducing the bugs. Size reports the lines of code before preprocessing. Δ reports the percentage of changed lines of code across versions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Size (KLOC)</th>
<th>Δ</th>
<th>#Bugs</th>
<th>Bug Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Old</td>
<td>New</td>
<td>Old</td>
<td>New</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shntool</td>
<td>3.0.4</td>
<td>3.0.5</td>
<td>13</td>
<td>13</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>latex2rtf</td>
<td>2.1.0</td>
<td>2.1.1</td>
<td>27</td>
<td>27</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>urjtag</td>
<td>0.7</td>
<td>0.8</td>
<td>45</td>
<td>46</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>optipng</td>
<td>0.5.2</td>
<td>0.5.3</td>
<td>60</td>
<td>61</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>wget</td>
<td>1.11.4</td>
<td>1.12</td>
<td>42</td>
<td>65</td>
<td>47</td>
<td>6</td>
</tr>
<tr>
<td>readeLF</td>
<td>2.23.2</td>
<td>2.24</td>
<td>63</td>
<td>65</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>grep</td>
<td>2.18</td>
<td>2.19</td>
<td>68</td>
<td>68</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>sed</td>
<td>4.2.2</td>
<td>4.3</td>
<td>48</td>
<td>83</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>sort</td>
<td>7.1</td>
<td>7.2</td>
<td>96</td>
<td>98</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>tar</td>
<td>1.27</td>
<td>1.28</td>
<td>108</td>
<td>112</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2. Effectiveness of Drake. Batch reports the number of alarms in each program version. Bingo and SynMask show the results of the baselines: the number of interactions until all bugs have been discovered, and the number of highlighted alarms and missed bugs respectively. DrakeUnsound and DrakeSound show the performance of Drake in each setting.

<table>
<thead>
<tr>
<th>Program</th>
<th>Batch</th>
<th>Bingo</th>
<th>SynMask</th>
<th>DrakeUnsound</th>
<th>DrakeSound</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Old</td>
<td>#New</td>
<td>#Iters</td>
<td>#Missed</td>
<td>#Diff</td>
<td>Initial</td>
</tr>
<tr>
<td>shntool</td>
<td>20</td>
<td>23</td>
<td>13</td>
<td>3</td>
<td>N/A</td>
</tr>
<tr>
<td>latex2rtf</td>
<td>7</td>
<td>13</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>urjtag</td>
<td>15</td>
<td>35</td>
<td>22</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>optipng</td>
<td>50</td>
<td>67</td>
<td>14</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>wget</td>
<td>850</td>
<td>793</td>
<td>168</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>readeLF</td>
<td>841</td>
<td>882</td>
<td>80</td>
<td>0</td>
<td>108</td>
</tr>
<tr>
<td>grep</td>
<td>916</td>
<td>913</td>
<td>53</td>
<td>1</td>
<td>204</td>
</tr>
<tr>
<td>sed</td>
<td>572</td>
<td>818</td>
<td>102</td>
<td>0</td>
<td>398</td>
</tr>
<tr>
<td>sort</td>
<td>684</td>
<td>715</td>
<td>177</td>
<td>0</td>
<td>41</td>
</tr>
<tr>
<td>tar</td>
<td>1,229</td>
<td>1,369</td>
<td>219</td>
<td>0</td>
<td>156</td>
</tr>
<tr>
<td>Total</td>
<td>5,184</td>
<td>5,628</td>
<td>854</td>
<td>4</td>
<td>1,178</td>
</tr>
</tbody>
</table>
DRAKE_{Unsound} may miss real bugs: in cases where this occurs, we mark the field as N/A.

In general, the number of alarms of the batch-mode analyses (the “Batch” columns) are proportional to the size of the program. Likewise, the number of syntactically new alarms by SYNMask is proportional to the amount of syntactic difference. Counterintuitive examples are wget, grep, and readelf. In case of wget, the number of alarms decreased even though the code size increased. It is mainly because a part of user-defined functionalities which reported many alarms has been replaced with library calls. Furthermore, a large part of the newly added code consists of simple wrappers of library calls that do not have buffer accesses. On the other hand, small changes of grep and readelf introduced many new alarms because the changes are mostly in core functionalities that heavily use buffer accesses. When such a complex code change happens, SYNMask cannot suppress false alarms effectively and can even miss real bugs. In case of grep, SYNMask still reports 22.3% of alarms compared to the batch mode and misses the newly introduced bug.

On the other hand, DRAKE consistently shows effectiveness in the various cases. For example, DRAKE_{Unsound} initially shows the bug in readelf at rank 28, and this ranking rises to 4 after transferring the old feedback. Finally the bug is presented at the top only within 4 iterations out of 108 syntactically new alarms. Furthermore, DRAKE_{Sound} requires only 9 iterations to detect the bug in grep that is missed by the syntactic approach, which was initially ranked at 15. In some benchmarks, such as shntool and tar, the rankings sometimes become worse after feedback. For example, the last true alarm of tar drops from its initial rank of 56 to 82 after feedback transfer. Observe that, in these cases, the number of alarms is either small (shntool), or the initial ranking is already very good (tar). Therefore, small amounts of noise in these benchmarks can result in a few additional iterations to discover all real bugs. This phenomenon occurs because of false generalization from user feedback, which in turn results from various sources of imprecision including abstract semantics, approximate derivation graphs, or approximate marginal inference. However, interactive reprioritization gradually improves the quality of the ranking, and the bug is eventually found within 32 rounds of feedback out of a total 1,369 alarms reported in the new version.

In total, DRAKE dramatically reduces manual effort for inspecting alarms. The original analysis in the batch mode reports 5,184 and 5,628 alarms for old and new versions of programs, respectively. Applying BINGO on the alarms from new versions requires the user to inspect 854 (15.2%) alarms. SYNMask suppresses all the previous alarms and reports 1,178 (20.9%) alarms. However, SYNMask misses 4 bugs that were previously false alarms in the old version. DRAKE_{Unsound} misses the same 4 bugs because it also suppresses the old alarms. Instead, DRAKE_{Unsound} presents the remaining bugs only within 171 (3.0%) iterations. DRAKE_{Sound} finds all the bugs within 299 (5.3%) iterations, a significant improvement over the baseline approaches.

### 5.3 Sensitivity analysis on different configurations

This section conducts a sensitivity study with different values of parameter \( \epsilon \) for DRAKE_{Sound}. Recall that \( \epsilon \) represents the degree of belief that the same abstract derivation tree from two versions has different concrete behaviors. Therefore, the higher \( \epsilon \) is set, the more conservatively DRAKE behaves.

Figure 7 shows the normalized number of iterations until all the bugs have been found by DRAKE_{Sound} with different values for \( \epsilon \). We observe that the overall number of iterations generally increases as \( \epsilon \) increases because DRAKE_{Sound} conservatively suppresses the old information. However, the rankings move opposite to this trend in some cases such as latex2rtf, readelf, and tar. In practice, various kinds of factors are involved in the probability of each alarm such
The reports are typically based on syntactic features of the behavior, and subsequently verify that the new version is bug-free under the same environment assumptions. Therefore, these approaches usually need general-purpose program verifiers, significant manual annotations, and do not consider the problems of user interaction or alarm ranking.

Research on hyperproperties [9] and on relational verification [3] relates the behaviors of a single program on multiple inputs or of multiple programs on the same input. Typical problems studied include equivalence checking [28, 34, 51, 54], information flow security [47], and verifying the correctness of code transformations [27]. Various logical formulations, such as Hoare-style partial equivalence [17], and techniques such as differential symbolic execution [52, 54] have been explored. In contrast to our work, such systems focus on identifying divergent behaviors between programs. On the other hand, in our case, it is almost certain that the programs are semantically inequivalent, and our focus is instead on differential bug-finding.

Finally, there is a large body of research leveraging probabilistic methods and machine learning to improve static analysis accuracy [26, 30, 32, 37, 38] and find bugs in programs [33, 39]. The idea of using Bayesian inference for interactive alarm prioritization which figures prominently in Drake follows our recent work on Bingo [53]. However, the main technical contribution of the present paper is the concept of semantic alarm masking which is enabled by the syntactic matching function and the differential derivation graph. This allows us to prioritize alarms that are relevant to the current code change. Orthogonally, when integrated with Bingo, the differential derivation graph also allows for generalization from user feedback, and transferring this feedback across multiple program versions. To the best of our knowledge, our work is the first to apply such techniques to reasoning about continuously evolving programs.

### 7 Conclusion

We have presented a system, Drake, for the analysis of continuously evolving programs. Drake prioritizes alarms according to their likely relevance relative to the last code change, and reranks alarms in response to user feedback. Drake operates by comparing the results of the static analysis runs from each version of the program, and builds a probabilistic model of alarm relevance using a differential derivation graph. Our experiments on a suite of ten widely-used C programs demonstrate that Drake dramatically reduces the alarm inspection burden compared to other state-of-the-art techniques without missing any bugs.

### Acknowledgments

We thank the anonymous reviewers and our shepherd, Sasa Misailovic, for insightful comments. This research was supported by DARPA under agreement #FA8750-15-2-0009, by NSF awards #1253867 and #1526270, and by a Facebook Research Award.
References


