Continuously Reasoning about Programs using Differential Bayesian Inference

Kihong Heo*
University of Pennsylvania, USA
kheo@cis.upenn.edu

Mukund Raghothaman*
University of Pennsylvania, USA
rmukund@cis.upenn.edu

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

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

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**.

*The first two authors contributed equally to this work.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314616

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

ACM Reference Format:
Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously Reasoning about Programs using Differential Bayesian Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. 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* of alarms—logical chains of cause-and-effect—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 to not only 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 to further 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-overrun 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?*

```
1 - #define CMP_SIZE 529200
2 #define HEADER_SIZE 44
3 + int shift_secs;
4
5 void read_value_long(FILE *file, long *val) {
6     char buf[5];
7     fread(buf, 1, 4, file); // Input Source
8     buf[4] = 0;
9     *val = (buf[3] << 24) | (buf[2] << 16) | (buf[1] << 8) | buf[0];
10 }
11
12 wave_info *new_wave_info(char *filename) {
13     wave_info *info;
14     FILE *f;
15
16     info = malloc(sizeof(wave_info));
17     f = fopen(filename);
18     read_value_long(f, info->header_size);
19     read_value_long(f, info->data_size);
20     return info;
21 }
22
23 void trim_main(char *filename) {
24     wave_info *info;
25     info = new_wave_info(filename);
26     long header_size;
27     char *header;
28
29     header_size = min(info->header_size, HEADER_SIZE);
30     header = malloc(header_size * sizeof(char)); // Alarm(30)
31     /* trim a wave file */
32 }
```

```
33 void cmp_main(char *filename1, char *filename2) {
34     wave_info *info1, *info2;
35     long bytes;
36     char *buf;
37
38     info1 = new_wave_info(filename1);
39     info2 = new_wave_info(filename2);
40
41 -   bytes = min(min(info1->data_size, info2->data_size), CMP_SIZE);
42 +   cmp_size = shift_secs * info1->rate; // Integer Overflow
43 +   bytes = min(min(info1->data_size, info2->data_size), cmp_size);
44
45     buf = malloc(2 * bytes * sizeof(char)); // Alarm(45)
46     /* compare two wave files */
47 }
48
49 int main(int argc, char *argv) {
50     int c ;
51     while ((c = getopt(argc, argv, "c:f:ls")) != -1) {
52         switch (c) {
53         case 'c':
54 +           shift_secs = atoi(optarg); // Input Source
55             cmp_main(argv[optind], argv[optind + 1]);
56             break;
57         case 't':
58             trim_main(argv[optind]);
59             break;
60     }
61     return 0;
62 }
```

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.

Figure 2 presents an overview of our system, Drake. First, the system extracts static analysis results from both the old and new versions of the program. Since these results are described in terms of syntactic entities (such as source locations) from different versions of the program, it uses a syntactic matching function δ to translate the old version of the constraints into the setting of the new program. Drake then merges the two sets of constraints into a unified *differential derivation graph*. These differential derivations highlight the relevance of the changed code to the static analysis alarms. Moreover, the differential derivation graph also enables us to perform marginal inference with the feedback from the user as well as previously labeled alarms from the old version.

We briefly explain the reasoning performed by Sparrow in Section 2.1, and explain our ideas in Sections 2.2–2.3.

2.1 Reflecting on the Integer Overflow Analysis

Sparrow detects harmful integer overflows by performing a flow-, field-, and context-sensitive taint analysis from untrusted data sources to sensitive sinks [21]. While the actual implementation includes complex details to ensure performance and accuracy, it can be approximated by inference rules such as those shown in Figure 3.

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 r_2, DUPath(c_1, c_3) :– DUPath(c_1, c_2), DUEdge(c_2, c_3), indicates that for each triple (c_1, c_2, c_3) of program points, whenever there is a multi-step data flow from c_1 to c_2 and an immediate data flow from c_2 to c_3, there may be a multi-step data flow from c_1 to c_3. 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

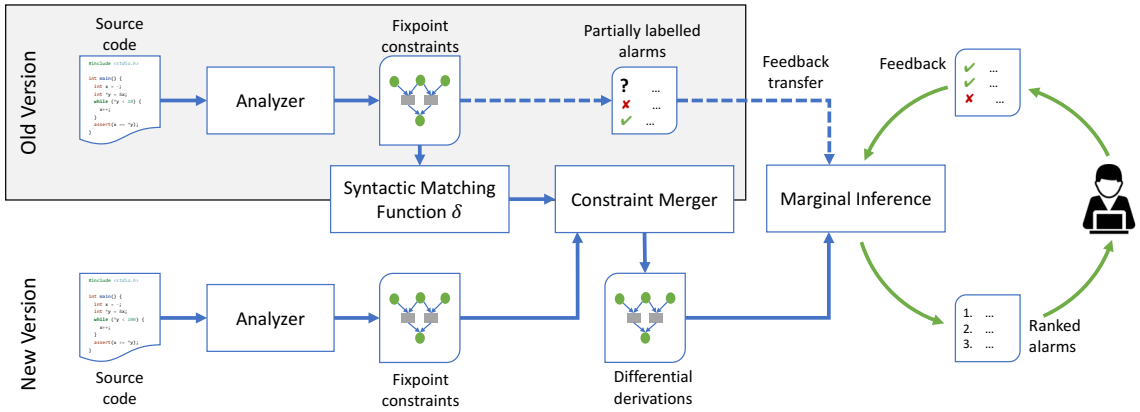


Figure 2. The DRAKE system. By applying the analysis to each version of the program, we start with the grounded constraints for each version. The syntactic matching function δ allows us to compare the constraints and derivation trees of P_{old} and P_{new}, which we merge to obtain the differential derivation graph GC_{Δ}. We treat the resulting structure as a probabilistic model, and interactively reprioritize the list of alarms as the user triages them and labels their ground truth.

Input relations	
$\text{DUEdge}(c_1, c_2)$:	Immediate data flow from c_1 to c_2
$\text{Src}(c)$:	Origin of potentially erroneous traces
$\text{Dst}(c)$:	Potential program crash point
Output relations	
$\text{DUPath}(c_1, c_2)$:	Transitive data flow from c_1 to c_2
$\text{Alarm}(c)$:	Potentially erroneous trace reaching c
Analysis rules	
r_1 :	$\text{DUPath}(c_1, c_2) \coloneq \text{DUEdge}(c_1, c_2)$.
r_2 :	$\text{DUPath}(c_1, c_3) \coloneq \text{DUPath}(c_1, c_2), \text{DUEdge}(c_2, c_3)$.
r_3 :	$\text{Alarm}(c_2) \coloneq \text{DUPath}(c_1, c_2), \text{Src}(c_1), \text{Dst}(c_2)$.

Figure 3. Approximating a complex taint analysis with simple inference rules. All variables c_1, c_2, etc. range over the set of program points.

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* δ 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 `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 $t_1 \wedge t_2 \wedge \cdots \wedge t_k \implies_r t$ together with a derivation tree τ_i for each antecedent tuple t_i.

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

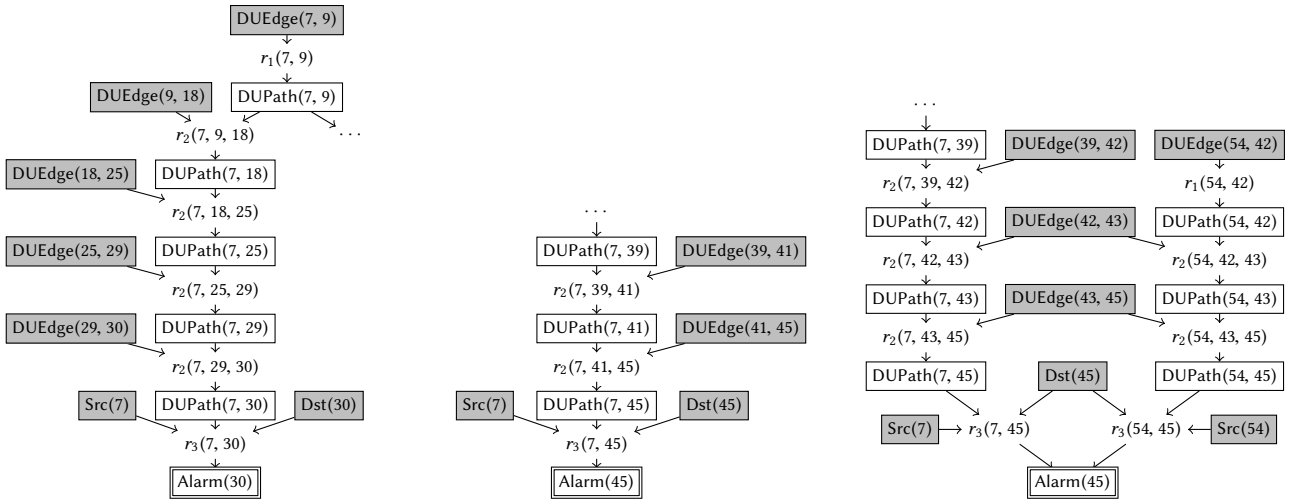
$$\text{DUPath}(7, 9) \rightarrow \text{DUPath}(7, 18) \rightarrow \cdots \rightarrow \text{Alarm}(30),$$

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

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

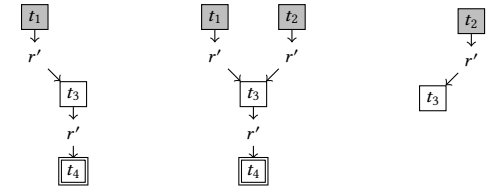
and where each sequence is supplemented with appropriate input tuples. Observe that each clause of the first tree, τ_{30}, is common to both P_{old} and P_{new}. More generally, *every* derivation tree of Alarm(30) from P_{new} is *already present* in P_{old}. As a result, Alarm(30) is unlikely to represent a real bug. On the other hand, the second tree, τ_{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



(a) Derivation tree common to P_{old} and P_{new} **(b)** Derivation tree exclusive to P_{old} **(c)** Derivation trees exclusive to P_{new}

Figure 4. Portions of the old and new derivation graphs by which the analysis identifies suspicious source-sink flows in the two versions of the program. The numbers indicate line numbers of the corresponding code in Figure 1. Nodes corresponding to grounded clauses, such as $r_1(7, 9)$, indicate the name of the rule and the instantiation of its variables, i.e., r_1 with $c_1 = 7$ and $c_2 = 9$. Notice that in the new derivation graph the analysis has discovered two suspicious flows—from lines 7 and 54 respectively—which both terminate at line 45.



(a) GC_{old} **(b)** GC_{new} **(c)** $GC_{\text{new}} \setminus GC_{\text{old}}$

Figure 5. Deleting clauses common to both versions—$t_1 \to t_3$ and $t_3 \to t_4$—hides the presence of a new derivation tree leading to t_4: $t_2 \to t_3 \to t_4$. Naive "local" approaches, based on tree or graph differences, are therefore insufficient to determine alarms which possess new derivation trees.

transitively extends 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.

The differential derivation graph. Notice that a derivation tree τ is either an input tuple t or a grounded clause $t_1 \wedge t_2 \wedge \cdots \wedge t_k \Longrightarrow_r t$ applied to a set of smaller derivation trees $\tau_1, \tau_2, \ldots, \tau_k$. If τ is an input tuple, then it is exclusive to the new analysis run iff it does not appear in the old program. In the inductive case, τ is exclusive to the new version iff, for some i, the sub-derivation τ_i is in turn exclusive to P_{new}.

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

$$g = \text{DUPath}(7, 9) \wedge \text{DUEdge}(9, 18) \Longrightarrow_{r_2} \text{DUPath}(7, 18). \quad (1)$$

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

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

$$g' = \text{DUPath}(7, 39) \wedge \text{DUEdge}(39, 42) \Longrightarrow_{r_2} \text{DUPath}(7, 42), \quad (2)$$

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

Our key insight is that we can perform this classification of derivation trees by splitting each tuple t into two variants, t_α and t_β. We set this up so that the derivations of t_α correspond exactly to the trees which are common to both versions, and the derivations of t_β correspond exactly to the trees which are exclusive to P_{new}. For example, the clause g splits into four copies, $g_{\alpha\alpha}$, $g_{\alpha\beta}$, $g_{\beta\alpha}$ and $g_{\beta\beta}$, for each combination of

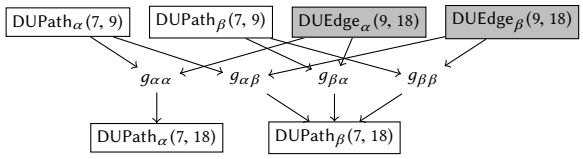


Figure 6. Differentiating the clause g from Equation 1.

antecedents:

$$g_{\alpha\alpha} = \mathsf{DUPath}_\alpha(7,9) \wedge \mathsf{DUEdge}_\alpha(9,18) \Longrightarrow_{r_2} \mathsf{DUPath}_\alpha(7,18), \quad (3)$$

$$g_{\alpha\beta} = \mathsf{DUPath}_\alpha(7,9) \wedge \mathsf{DUEdge}_\beta(9,18) \Longrightarrow_{r_2} \mathsf{DUPath}_\beta(7,18), \quad (4)$$

$$g_{\beta\alpha} = \mathsf{DUPath}_\beta(7,9) \wedge \mathsf{DUEdge}_\alpha(9,18) \Longrightarrow_{r_2} \mathsf{DUPath}_\beta(7,18), \text{ and} \quad (5)$$

$$g_{\beta\beta} = \mathsf{DUPath}_\beta(7,9) \wedge \mathsf{DUEdge}_\beta(9,18) \Longrightarrow_{r_2} \mathsf{DUPath}_\beta(7,18). \quad (6)$$

Observe that the only way to derive $\mathsf{DUPath}_\alpha(7,18)$ is by applying a clause to a set of tuples all of which are themselves of the α-variety. The use of even a single β-variant hypothesis always results in the production of $\mathsf{DUPath}_\beta(7,18)$. 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_Δ.

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

In contrast, the hypothesis DUEdge(39, 42) of g' is only present in P_{new}, so that we include $\mathsf{DUEdge}_\beta(39,42)$ in I_Δ, but exclude its α-variant. As a result, $g'_{\alpha\beta} = \mathsf{DUPath}_\alpha(7,39) \wedge \mathsf{DUEdge}_\beta(39,42) \Longrightarrow_{r_2} \mathsf{DUPath}_\beta(7,42)$ successfully fires, but all of its siblings—$g'_{\alpha\alpha}$, $g'_{\beta\alpha}$, and $g'_{\beta\beta}$—are inactive. The differential derivation graph, GC_Δ, thus enables the successful derivation of $\mathsf{DUPath}_\beta(7,42)$, and of all its consequences, $\mathsf{DUPath}_\beta(7,43)$, $\mathsf{DUPath}_\beta(7,45)$, and $\mathsf{Alarm}_\beta(45)$.

2.3 A Probabilistic Model of Alarm Relevance

We build our system on the idea of highlighting alarms $\mathsf{Alarm}(c)$ whose β-variants, $\mathsf{Alarm}_\beta(c)$, 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_2, and its instantiation as the grounded clause in Figure 6, $g_{\alpha\beta} = r_2(t_1, t_2)$, with $t_1 = \mathsf{DUPath}_\alpha(7,9)$ and $t_2 = \mathsf{DUEdge}_\beta(9,18)$ as its antecedent tuples, and with $t_3 = \mathsf{DUPath}_\beta(7,18)$ as its conclusion. We define:

$$\Pr(g_{\alpha\beta} \mid t_1 \wedge t_2) = p_{r_2}, \text{ and} \quad (7)$$

$$\Pr(g_{\alpha\beta} \mid \neg t_1 \vee \neg t_2) = 0, \quad (8)$$

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_2}.[1] The conclusion t_3 is true iff any one of its deriving clauses successfully fires:

$$\Pr(t_3 \mid g_{\alpha\beta} \vee g_{\beta\alpha} \vee g_{\beta\beta}) = 1, \text{ and} \quad (9)$$

$$\Pr(t_3 \mid \neg(g_{\alpha\beta} \vee g_{\beta\alpha} \vee g_{\beta\beta})) = 0. \quad (10)$$

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

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

Furthermore, $\boldsymbol{e}$ 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_Δ, i.e., to $\mathsf{Alarm}_\alpha(45)$. We note that this

[1] There 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.

combination of differential relevance computation and probabilistic generalization of feedback is dramatically effective in practice: while the original analysis produces an average of 563 alarms in each our benchmarks, after relevance-based ranking, the last real bug is at rank 94; the initial feedback transfer reduces this to rank 78, and through the process of interactive reranking, all true bugs are discovered within just 30 rounds of interaction on average.

3 A Framework for Alarm Transfer

We formally describe the Drake workflow in Algorithm 1, and devote this section to our core technical contributions: the constraint merging algorithm Merge in step 3 and enabling feedback transfer in step 5. We begin by setting up preliminary details regarding the analysis and reviewing the use of Bayesian inference for interactive alarm ranking.

Algorithm 1 Drake$_{\mathcal{A}}(P_{\text{old}}, P_{\text{new}})$, where $\mathcal{A}$ is an analysis, and P_{old} and P_{new} are the old and new versions of the program to be analyzed.

1. Compute $\mathbf{R}_{\text{old}} = \mathcal{A}(P_{\text{old}})$ and $\mathbf{R}_{\text{new}} = \mathcal{A}(P_{\text{new}})$. Analyze both programs.
2. Define $\mathbf{R}_\delta = \delta(\mathbf{R}_{\text{old}})$. Translate the analysis results and feedback from P_{old} to the setting of P_{new}.
3. Compute the differential derivation graph:
$$\mathbf{R}_\Delta = \textsc{Merge}(\mathbf{R}_\delta, \mathbf{R}_{\text{new}}). \tag{11}$$
4. Pick a bias ϵ according to Section 3.2 and convert $\mathbf{R}_\Delta$ into a Bayesian network, Bnet($\mathbf{R}_\Delta$). Let Pr be its joint probability distribution.
5. Initialize the feedback set $\boldsymbol{e}$ according to the chosen feedback transfer mode (see Section 3.3).
6. While there exists an unlabelled alarm:
   a. Let A_u be the set of unlabelled alarms.
   b. Present the highest probability unlabelled alarm for user inspection:
   $$a = \arg\max_{a \in A_u} \Pr(a_\beta \mid \boldsymbol{e}).$$
   If the user marks it as true, update $\boldsymbol{e} := \boldsymbol{e} \wedge a_\beta$. Otherwise update $\boldsymbol{e} := \boldsymbol{e} \wedge \neg a_\beta$.

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 $R_h(\boldsymbol{v}_h) :- R_1(\boldsymbol{v}_1), R_2(\boldsymbol{v}_2), \ldots, R_k(\boldsymbol{v}_k)$, where $R_1, R_2, \ldots, R_k$ are relations, R_h is an output relation, $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k$ and $\boldsymbol{v}_h$ are vectors of variables of appropriate arity. The rule r encodes the following universally quantified logical formula: "For all values of $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k$ and $\boldsymbol{v}_h$, if $R_1(\boldsymbol{v}_1) \wedge R_2(\boldsymbol{v}_2) \wedge \cdots \wedge R_k(\boldsymbol{v}_k)$, then $R_h(\boldsymbol{v}_h)$."

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 $R_1(\boldsymbol{c}_1), R_2(\boldsymbol{c}_2), \ldots, R_k(\boldsymbol{c}_k) \in C$, we update $C := C \cup \{R_h(\boldsymbol{c}_h)\}$ and

$$GC := GC \cup \{R_1(\boldsymbol{c}_1) \wedge R_2(\boldsymbol{c}_2) \wedge \cdots \wedge R_k(\boldsymbol{c}_k) \Longrightarrow_r R_h(\boldsymbol{c}_h)\}.$$

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 [53] 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 $t_1 \wedge t_2 \wedge \cdots \wedge t_k \Longrightarrow_r t_h$. 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 $\boldsymbol{p}$ 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 $\mathcal{P}$:

$$\mathcal{P}(g \mid t_1 \wedge t_2 \wedge \cdots \wedge t_k) = p_r, \text{ and} \tag{12}$$
$$\mathcal{P}(g \mid \neg(t_1 \wedge t_2 \wedge \cdots \wedge t_k)) = 0. \tag{13}$$

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

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:

$$\mathcal{P}(t \mid g_1 \vee g_2 \vee \cdots \vee g_l) = 1, \text{ and} \tag{14}$$
$$\mathcal{P}(t \mid \neg(g_1 \vee g_2 \vee \cdots \vee g_l)) = 0. \tag{15}$$

Let us also assume a function $\boldsymbol{p}_{\text{in}}$ which maps input tuples t to their prior probabilities. In the simplest case, input tuples are known with certainty, so that $\boldsymbol{p}_{\text{in}}(t) = 1$. In Section 3.2, we will see that the choice of $\boldsymbol{p}_{\text{in}}$ allows us to uniformly generalize both relevance-based and traditional batch-mode ranking. We define the CPD of each input tuple t as:

$$\mathcal{P}(t) = \boldsymbol{p}_{\text{in}}(t). \tag{16}$$

By definition, a Bayesian network is a pair $(\mathcal{G}, \mathcal{P})$, where $\mathcal{G}$ is an acyclic graph and $\mathcal{P}$ is an assignment of CPDs to each node [31]. We have already defined the CPDs in Equations 12–16; the challenge is that the derivation graph G may have cycles. Raghothaman et al. [53] 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, $\textsc{Bnet}(\mathbf{R}) = (G_c, \mathcal{P})$.

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* $\mathbf{R}_\Delta$. Every derivation tree τ of a tuple from $\mathbf{R}_{\text{new}}$ is either common to both $\mathbf{R}_\delta$ and $\mathbf{R}_{\text{new}}$, 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 $t_1 \wedge t_2 \wedge \cdots \wedge t_k \implies_r t_h$ together with derivation trees $\tau_1, \tau_2, \ldots, \tau_k$ for each of the antecedent tuples. Since the grounded clauses are collected until fixpoint, the only way for a derivation tree to be exclusive to the new program is if it is either: (*a*) a new input tuple $t \in I_{\text{new}} \setminus I_\delta$, or (*b*) a clause $t_1 \wedge t_2 \wedge \cdots \wedge t_k \implies_r t_h$ with a new derivation tree for at least one child t_i.

The idea behind the construction of $\mathbf{R}_\Delta$ is therefore to split each tuple t into two *variants*, t_α and t_β, where t_α precisely captures the common derivation trees and t_β exactly captures the derivation trees which only occur in $\mathbf{R}_{\text{new}}$. 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 $\boldsymbol{R}_\Delta = \textsc{Merge}(\boldsymbol{R}_\delta, \boldsymbol{R}_{new})$. Then, for each tuple t,*

1. *t_α is derivable from $\boldsymbol{R}_\Delta$ iff t has a derivation tree which is common to both $\boldsymbol{R}_\delta$ and $\boldsymbol{R}_{new}$, and*
2. *t_β is derivable from $\boldsymbol{R}_\Delta$ iff t has a derivation tree which is absent from $\boldsymbol{R}_\delta$ but present in $\boldsymbol{R}_{new}$.*

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 $\Rightarrow$ direction. Let t_α be the result of a clause $t'_1 \wedge t'_2 \wedge \cdots \wedge t'_k \implies_r t_\alpha$. By construction, it is the case that each t'_i is of the form $t_{i\alpha}$, and by IH, it must already have a derivation tree τ_i which is common to both analysis results. It follows that t_α also has a derivation tree $r(\tau_1, \tau_2, \ldots, \tau_k)$ in common to both results.

Algorithm 2 $\textsc{Merge}(\mathbf{R}_\delta, \mathbf{R}_{\text{new}})$, where $\mathbf{R}_\delta$ is the translated analysis result from P_{old} and $\mathbf{R}_{\text{new}}$ is the result from P_{new}.

1. Unpack the input-, output-, alarm tuples, and grounded clauses from each version of the analysis result. Let $(I_\delta, C_\delta, A_\delta, GC_\delta) = \mathbf{R}_\delta$ and $(I_{\text{new}}, C_{\text{new}}, A_{\text{new}}, GC_{\text{new}}) = \mathbf{R}_{\text{new}}$.
2. Form two versions, t_α, t_β, of each output tuple in $\mathbf{R}_{\text{new}}$:
$$C_\Delta = \{t_\alpha, t_\beta \mid t \in C_{\text{new}}\}, \text{ and}$$
$$A_\Delta = \{t_\alpha, t_\beta \mid t \in A_{\text{new}}\}.$$
3. Classify the input tuples into those which are common to both versions and those which are exclusively new:
$$I_\Delta = \{t_\alpha \mid t \in I_{\text{new}} \cap I_\delta\} \cup \{t_\beta \mid t \in I_{\text{new}} \setminus I_\delta\}.$$
4. Populate the clauses of GC_Δ: For each clause $g \in GC_{\text{new}}$ of the form $t_1 \wedge t_2 \wedge \cdots \wedge t_k \implies_r t_h$, and for each $H'_g \in \{t_{1\alpha}, t_{1\beta}\} \times \{t_{2\alpha}, t_{2\beta}\} \times \cdots \times \{t_{k\alpha}, t_{k\beta}\}$,
   a. if $H'_g = (t_{1\alpha}, t_{2\alpha}, \ldots, t_{k\alpha})$ consists entirely of "α"-tuples, produce the clause:
   $$H'_g \implies_r t_{h\alpha}.$$
   b. Otherwise, if there is at least one "β"-tuple, then emit the clause:
   $$H'_g \implies_r t_{h\beta}.$$
5. Output the merged result $\mathbf{R}_\Delta = (I_\Delta, C_\Delta, A_\Delta, GC_\Delta)$.

In the $\Leftarrow$ direction. t is the result of a clause $t_1 \wedge t_2 \wedge \cdots \wedge t_k \implies_r t$, where each t_i has a derivation tree τ_i which is common to both versions. By IH, it follows that $t_{i\alpha}$ is derivable in $\mathbf{R}_\Delta$ for each i, and therefore that t_α is also derivable in the merged results.

Of part 2, in the $\Rightarrow$ direction. Let t_β be the result of a clause $t'_1 \wedge t'_2 \wedge \cdots \wedge t'_k \implies_r t_\beta$. By construction, $t'_i = t_{i\beta}$ for at least one i, so that t_i has an exclusively new derivation tree τ_i. For all $j \neq i$, so that $t'_j \in \{t_{j\alpha}, t_{j\beta}\}$, t_j has a derivation tree τ_j either by IH or by part 1. By combining the derivation trees τ_l for each $l \in \{1, 2, \ldots, k\}$, we obtain an exclusively new derivation tree $r(\tau_1, \tau_2, \ldots, \tau_l)$ which produces t.

In the $\Leftarrow$ direction. Let the exclusively new derivation tree τ of t be an instance of the clause $t_1 \wedge t_2 \wedge \ldots t_k \implies t$, and let τ_i be one sub-tree which is exclusively new. By IH, it follows that $t_{i\beta}$, and that therefore, t_β are both derivable in $\mathbf{R}_\Delta$. □

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 $k_{\max}$ is the size of the largest rule body, then the algorithm runs in $O(2^{k_{\max}}|\mathbf{R}_{\text{new}}|)$ time and produces $\mathbf{R}_\Delta$ which is also of size $O(2^{k_{\max}}|\mathbf{R}_{\text{new}}|)$. Given a tuple $t \in C_{\text{new}}$, the existence of a derivation tree exclusive to $\mathbf{R}_{\text{new}}$ can be determined using Theorem 3.1 in time $O(|\mathbf{R}_\Delta|)$. In practice, since the analysis is fixed with $k_{\max} < 4$, these

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_{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 $\mathbf{R}_\Delta$. Therefore, instead of using I_Δ as the set of input tuples in $\textsc{Bnet}(\mathbf{R}_\Delta)$, 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 $\boldsymbol{p}_{\text{in}}(t_\beta) = 1 - \boldsymbol{p}_{\text{in}}(t_\alpha) = 1.0$, and otherwise, if $t \in I_{\text{new}} \cap I_\delta$, then $\boldsymbol{p}_{\text{in}}(t_\beta) = 1 - \boldsymbol{p}_{\text{in}}(t_\alpha) = \epsilon$. Here, ϵ is our belief that the same abstract state has new concrete behaviors. The choice of ϵ 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 $\boldsymbol{e}_{\text{old}}$ 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 $\neg a \in \boldsymbol{e}_{\text{old}}$. The programmer has therefore indicated that *all of its derivation trees* in $\mathbf{R}_{\text{old}}$ are false. If $a' = \delta(a)$, since the derivation trees of a'_α in $\mathbf{R}_\Delta$ correspond to a subset of the derivation trees of a in $\mathbf{R}_{\text{old}}$, we can additionally deprioritize these derivation trees by initializing:

$$\boldsymbol{e} \coloneqq \{\neg a_\alpha \mid \forall \text{ negative labels } \text{“}\neg a\text{”} \in \delta(\boldsymbol{e}_{\text{old}})\}. \tag{17}$$

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

$$\boldsymbol{e} \coloneqq \{\neg a_\alpha \mid \forall a \in A_\delta\}. \tag{18}$$

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:

$$\boldsymbol{e} \coloneqq \{\neg a_\alpha, \neg a_\beta \mid \forall a \in A_\delta\}. \tag{19}$$

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 δ 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 $(\mathbb{C}, \rightarrow, c_0)$ where $\mathbb{C}$ denotes the set of program points, $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ 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:

$$\mathbb{D} = \mathbb{C} \rightarrow \mathbb{S}.$$

An abstract state maps abstract locations to abstract values:

$$\mathbb{S} = \mathbb{L} \rightarrow \mathbb{V}.$$

The analysis produces alarms for each potentially erroneous program points.

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

$$c_0 \overset{l}{\rightsquigarrow} c_n = \exists [c_0, c_1, \ldots, c_n] \in \mathsf{Paths}, \exists l \in \mathbb{L}. \\ l \in \mathsf{D}(c_0) \cap \mathsf{U}(c_n) \wedge \forall i \in (0, n). l \notin \mathsf{D}(c_i)$$

where $\mathsf{D}(c) \subseteq \mathbb{L}$ and $\mathsf{U}(c) \subseteq \mathbb{L}$ denote the def and use sets of abstract locations at program point c. A data dependency $c_0 \overset{l}{\rightsquigarrow} c_n$ 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 $\mathsf{DUEdge} \subseteq \mathbb{C} \times \mathbb{C}$ which is a variant of data dependency [48] is defined as follows:

$$\mathsf{DUEdge}(c_0, c_n) = \exists l \in \mathbb{L}. c_0 \overset{l}{\rightsquigarrow} c_n.$$

A dataflow relation $\mathsf{DUEdge}(c_0, c_n)$ represents that an abstract location is defined at program point c_0 and used at c_n. Relation $\mathsf{DUPath}(c_1, c_n)$ represents transitive dataflow relation from point c_1 to c_n. Relation $\mathsf{Alarm}(c_1, c_n)$ 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 $\delta \in \mathbb{C}_{P_1} \to (\mathbb{C}_{P_1} \uplus \mathbb{C}_{P_2})$:

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

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 δ 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 a handful of alarms with non-zero probability, all of which can then be immediately resolved by the developer.

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 i7 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.

Program	Version		Size (KLOC)		Δ	#Bugs	Bug Type	Reference
	Old	New	Old	New	(%)			
shntool	3.0.4	3.0.5	13	13	1	6	Integer overflow	[21]
latex2rtf	2.1.0	2.1.1	27	27	3	2	Format string	[11]
urjtag	0.7	0.8	45	46	18	6	Format string	[21]
optipng	0.5.2	0.5.3	60	61	2	1	Integer overflow	[12]
wget	1.11.4	1.12	42	65	47	6	Buffer overrun	[55, 56]
readelf	2.23.2	2.24	63	65	6	1	Buffer overrun	[13]
grep	2.18	2.19	68	68	7	1	Buffer overrun	[10]
sed	4.2.2	4.3	48	83	40	1	Buffer overrun	[18]
sort	7.1	7.2	96	98	3	1	Buffer overrun	[15]
tar	1.27	1.28	108	112	4	1	Buffer overrun	[43]

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. **Drake**$_{Unsound}$ and **Drake**$_{Sound}$ show the performance of Drake in each setting.

Program	Batch		Bingo	SynMask		Drake$_{Unsound}$			Drake$_{Sound}$		
	#Old	#New	#Iters	#Missed	#Diff	Initial	Feedback	#Iters	Initial	Feedback	#Iters
shntool	20	23	13	3	3	N/A	N/A	N/A	8	21	19
latex2rtf	7	13	6	0	6	5	6	5	12	9	6
urjtag	15	35	22	0	27	25	16	18	28	25	21
optipng	50	67	14	0	17	11	5	4	26	5	9
wget	850	793	168	0	218	123	140	55	393	318	124
readelf	841	882	80	0	108	28	4	4	216	182	25
grep	916	913	53	1	204	N/A	N/A	N/A	15	10	9
sed	572	818	102	0	398	262	209	60	154	118	41
sort	684	715	177	0	41	14	14	10	33	9	13
tar	1,229	1,369	219	0	156	23	29	15	56	82	32
Total	5,184	5,628	854	4	1,178	491	423	171	941	779	299

the Bayesian network extracted only from the new version of the program. SynMask, on the other hand, performs differential reasoning using the syntactic matching algorithm described in Section 4.3. This represents the straightforward approach to estimating alarm relevance, and is commonly used in tools such as Facebook Infer [50].

5.2 Effectiveness

This section evaluates the effectiveness of Drake's ranking compared to the baseline systems. We instantiate Drake with two different settings, Drake$_{Sound}$ and Drake$_{Unsound}$ as described in Section 3.3. Drake$_{Sound}$ is bootstrapped by assuming the old variants of common alarms to be false (strong mode in Section 3.3) and its input parameter ϵ is set to 0.001. Drake$_{Unsound}$ aggressively deprioritizes the alarms by assuming both of the old and new variants of common alarms to be false (aggressive mode in Section 3.3), and setting ϵ to 0. For each setting, we measure three metrics: (*a*) the quality of the initial ranking based on the differential derivation graph, (*b*) the quality of ranking after transferring old feedback, and (*c*) the quality of the interactive ranking process. For Bingo, we show the number of user interactions on the alarms only from the new version. For SynMask, we report the number of alarms and missed bugs after syntactic masking.

Table 2 shows the performance of each system. The "**Initial**" and "**Feedback**" columns report the positions of last true alarm in the initial ranking before and after feedback transfer (corresponding to metrics (*a*) and (*b*) above). In each step, the user inspects the top-ranked alarm, and we rerank the remaining alarms according to their feedback. The "**#Iters**" columns report the number of iterations after which all bugs were discovered (metric (*c*)). Recall that both SynMask and

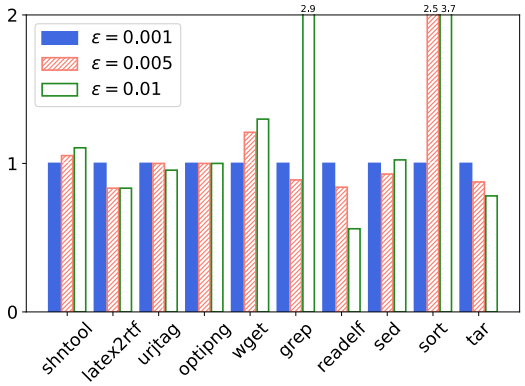


Figure 7. The normalized number of iterations until the last true alarm has been discovered with different values of parameter ϵ for DRAKE$_{Sound}$.

Table 3. Sizes of the old, new and merged Bayesian networks in terms of the number of tuples (**#T**) and clauses (**#C**), and the average iteration time on the merged network.

	Old		New		Merged		
Program	**#T**	**#C**	**#T**	**#C**	**#T**	**#C**	**Time(s)**
shntool	208	296	236	341	924	1,860	21
latex2rtf	152	179	710	943	1,876	3,130	17
urjtag	547	765	676	920	1,473	2,275	23
optipng	492	561	633	730	1,905	3,325	7
wget	3,959	4,484	3,297	3,608	9,264	14,549	23
grep	4,265	4,802	4,346	4,901	10,703	16,677	31
readelf	3702	4283	3,952	4,565	10,978	17,404	31
sed	1,887	2,030	2,971	3,265	6,914	9,998	15
sort	2,672	2,951	2,796	3,085	8,667	14,545	31
tar	5,620	6,197	6,096	6,708	18,118	30,252	47
Total	23,504	26,548	25,713	29,066	70,822	114,015	246

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 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 ϵ for DRAKE$_{Sound}$. Recall that ϵ represents the degree of belief that the same abstract derivation tree from two versions has different concrete behaviors. Therefore, the higher ϵ 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 ϵ. We observe that the overall number of iterations generally increases as ϵ 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

as structure of the network. For example, when bugs are closely related to many false alarms that were transformed from the old versions, an aggressive approach (i.e., small ϵ) can introduce negative effects. In fact, the bugs in the three benchmarks are closely related to huge functions or recursive calls that hinder precise static analysis. In such cases, aggressive assumptions on the previous derivations can be harmful for the ranking.

5.4 Scalability

The scalability of the iterative ranking process mostly depends on the size of the Bayesian network. Drake optimizes the Bayesian networks using optimization techniques described in previous work [53]. We measure the network size in terms of the number of tuples and clauses in derivation trees after the optimizations, and report the average time for each marginal inference computation where ϵ is set to 0.001.

Table 3 show the size and average computation time for each iteration. The merged networks have 3x more tuples and 4x more clauses compared to the old and new versions of networks. The average iteration time for all benchmarks is less than 1 minute which is reasonable for user interaction.

6 Related Work

Our work is inspired by recent industrial scale deployments of program analysis tools such as Coverity [4], Facebook Infer [50], Google Tricorder [57], and SonarQube [8]. These tools primarily employ syntactic masking to suppress reporting alarms that are likely irrelevant to a particular code commit. Indeed, syntactic program differencing goes back to the classic Unix `diff` algorithm proposed by Hunt and McIlroy in 1976 [23]. Our work builds upon these works and uses syntactic matching to identify abstract states before and after a code commit.

Program differencing techniques have been developed by the software engineering community [24, 29, 62]. Their goal is to summarize, to a human developer, the semantic code changes using dependency analysis or logical rules. The reports are typically based on syntactic features of the code change. On the other hand, our goal is to identify newly introduced bugs, and Drake captures deep semantic changes indicated by the program analysis in the derivation graph.

The idea of checking program properties using information obtained from its previous versions has also been studied by the program verification community, as the problem of differential static analysis [36]. Differential assertion checking [35], verification modulo versions [41], and the SymDiff project [20] are prominent examples of research in this area. The SafeMerge system [60] considers the problem of detecting bugs introduced while merging code changes. These systems typically analyze the old version of the program to obtain the environment conditions that preclude buggy 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

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1994. *Foundations of Databases: The Logical Level* (1st ed.). Pearson.

[2] Thomas Ball and Sriram Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*. ACM, 1–3.

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *Formal Methods (FM 2011)*. Springer, 200–214.

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75.

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*. ACM, 196–207.

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2009)*. ACM, 243–262.

[7] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Method Symposium*. Springer, 3–11.

[8] Ann Campbell and Patroklos Papapetrou. 2013. *SonarQube in Action* (1st ed.). Manning Publications Co.

[9] Michael Clarkson and Fred Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (Sept. 2010), 1157–1210.

[10] MITRE Corporation. 2015. CVE-2015-1345. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1345.

[11] MITRE Corporation. 2015. CVE-2015-8106. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8106.

[12] MITRE Corporation. 2017. CVE-2017-16938. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16938.

[13] MITRE Corporation. 2018. CVE-2018-10372. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10372.

[14] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1977)*. ACM, 238–252.

[15] Paul Eggert. 2010. `sort`: Commit `14ad7a2`. http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=14ad7a2. `sort`: Fix very-unlikely buffer overrun when merging to input file.

[16] Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *Proceedings of the International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*. Springer, 10–30.

[17] Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proceedings of the 46th Annual Design Automation Conference (DAC 2009)*. ACM, 466–471.

[18] Assaf Gordon. 2018. `sed`: Commit `007a417`. http://git.savannah.gnu.org/cgit/sed.git/commit/?id=007a417. `sed`: Fix heap buffer overflow from multiline EOL regex optimization.

[19] GrammaTech. 2005. CodeSonar. https://www.grammatech.com/products/codesonar.

[20] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards Modularly Comparing Programs Using Automated Theorem Provers. In *Proceedings of the International Conference on Automated Deduction (CADE 24)*. Springer, 282–299.

[21] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided Selectively Unsound Static Analysis. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. IEEE Press, 519–529.

[22] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Notices* 39, OOPSLA (Dec. 2004), 92–106.

[23] James Hunt and Douglas McIlroy. 1976. *An Algorithm for Differential File Comparison*. Technical Report. Bell Laboratories.

[24] Daniel Jackson and David Ladd. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proceedings of the International Conference on Software Maintenance (ICSM 1994)*. 243–252.

[25] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Proceedings of the International Conference on Computer Aided Verification (CAV 2016)*. Springer, 422–430.

[26] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. 2005. Taming False Alarms From a Domain-unaware C Analyzer by a Bayesian Statistical Post Analysis. In *Static Analysis: 12th International Symposium (SAS 2005)*. Springer, 203–217.

[27] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 631–645.

[28] Ming Kawaguchi, Shuvendu Lahiri, and Henrique Rebelo. 2010. *Conditional Equivalence*. Technical Report. Microsoft Research. https://www.microsoft.com/en-us/research/publication/conditional-equivalence/

[29] Miryung Kim and David Notkin. 2009. Discovering and Representing Systematic Code Changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society, 309–319.

[30] Ugur Koc, Parsa Saadatpanah, Jeffrey Foster, and Adam Porter. 2017. 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)*. ACM, 35–42.

[31] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.

[32] Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Static Analysis: 10th International Symposium (SAS 2003)*. Springer, 295–315.

[33] Ted Kremenek, Andrew Ng, and Dawson Engler. 2007. A Factor Graph Model for Software Bug Finding. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI 2007)*. Morgan Kaufmann, 2510–2516.

[34] Shuvendu Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SymDiff: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the International Conference on Computer Aided Verification (CAV 2012)*. Springer, 712–717.

[35] Shuvendu Lahiri, Kenneth McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, 345–355.

[36] Shuvendu Lahiri, Kapil Vaswani, and C. A. R. Hoare. 2010. Differential Static Analysis: Opportunities, Applications, and Challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010)*. ACM, 201–204.

[37] Wei Le and Mary Lou Soffa. 2010. Path-based Fault Correlations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010)*. ACM, 307–316.

[38] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012. Sound Non-statistical Clustering of Static Analysis Alarms. In *Verification, Model Checking, and Abstract Interpretation: 13th International Conference (VMCAI 2012)*. Springer, 299–314.

[39] Benjamin Livshits, Aditya Nori, Sriram Rajamani, and Anindya Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*. ACM, 75–86.

[40] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46.

[41] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. 2014. Verification Modulo Versions: Towards Usable Verification. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*. ACM, 294–304.

[42] Magnus Madsen and Anders Møller. 2014. Sparse Dataflow Analysis with Pointers and Reachability. In *Static Analysis*. Springer, 201–218.

[43] Jim Meyering. 2018. `tar`: Commit `b531801`. http://git.savannah.gnu.org/cgit/tar.git/commit/?id=b531801. One-top-level: Avoid a heap-buffer-overflow.

[44] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 7:1–7:24.

[45] Joris Mooij. 2010. libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models. *Journal of Machine Learning Research* 11 (Aug. 2010), 2169–2173.

[46] Mayur Naik. 2006. Chord: A Program Analysis Platform for Java. https://github.com/pag-lab/jchord.

[47] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2011. Verification of Information Flow and Access Control Policies with Dependent Types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP 2011)*. IEEE Computer Society, 165–179.

[48] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM, 229–238.

[49] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. The Sparrow static analyzer. https://github.com/ropas/sparrow.

[50] Peter O'Hearn. 2018. Continuous Reasoning: Scaling the Impact of Formal Methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2018)*. ACM, 13–25.

[51] Nimrod Partush and Eran Yahav. 2013. Abstract Semantic Differencing for Numerical Programs. In *Proceedings of the International Static Analysis Symposium (SAS 2013)*. Springer, 238–258.

[52] Suzette Person, Matthew Dwyer, Sebastian Elbaum, and Corina Păsăreanu. 2008. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008)*. ACM, 226–237.

[53] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 722–735.

[54] David Ramos and Dawson Engler. 2011. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the International Conference on Computer Aided Verification (CAV 2011)*. Springer, 669–685.

[55] Tim Rühsen. 2018. `wget`: Commit `b3ff8ce`. http://git.savannah.gnu.org/cgit/wget.git/commit/?id=b3ff8ce. `src/ftp-ls.c` (`ftp_parse_vms_ls`): Fix heap-buffer-overflow.

[56] Tim Rühsen. 2018. `wget`: Commit `f0d715b`. http://git.savannah.gnu.org/cgit/wget.git/commit/?id=f0d715b. `src/ftp-ls.c` (`ftp_parse_vms_ls`): Fix heap-buffer-overflow.

[57] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66.

[58] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 693–706.

[59] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*. ACM, 485–495.

[60] Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. 2018. Verified Three-way Program Merge. 2, OOPSLA (2018), 165:1–165:29.

[61] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, 265–266.

[62] Chungha Sung, Shuvendu Lahiri, Constantin Enea, and Chao Wang. 2018. Datalog-based Scalable Semantic Diffing of Concurrent Programs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, 656–666.