Boosting Static Analysis Accuracy with Instrumented Test Executions

Tianyi Chen
tianyichen@alumni.usc.edu
University of Southern California
USA

Kihong Heo
kihong.heo@kaist.ac.kr
KAIST
Korea

Mukund Raghothaman
raghotha@usc.edu
University of Southern California
USA

ABSTRACT

The two broad approaches to discover properties of programs—static and dynamic analyses—have complementary strengths: static techniques perform exhaustive exploration and prove upper bounds on program behaviors, while the dynamic analysis of test cases provides concrete evidence of these behaviors and promise low false alarm rates. In this paper, we present DynaBoost, a system which uses information obtained from test executions to prioritize the alarms of a static analyzer. We instrument the program to dynamically look for dataflow behaviors predicted by the static analyzer, and use these results to bootstrap a probabilistic alarm ranking system, where the user repeatedly inspects the alarm judged most likely to be a real bug, and where the system re-ranks the remaining alarms in response to user feedback. The combined system is able to exploit information that cannot be easily provided by users, and provides significant improvements in the human alarm inspection burden: by 35% compared to the baseline ranking system, and by 89% compared to an unaided programmer triaging alarm reports.

CCS CONCEPTS

• Software and its engineering → Automated static analysis; Dynamic analysis; • Mathematics of computing → Bayesian networks; • Information systems → Probabilistic retrieval models.

KEYWORDS

Static analysis, dynamic analysis, belief networks, Bayesian inference, alarm ranking

ACM Reference Format:
Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting Static Analysis Accuracy with Instrumented Test Executions. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468626

1 INTRODUCTION

Both static and dynamic analysis techniques have established themselves as important and complementary approaches to determine facts about programs. On the one hand, static analyses provide exhaustive validation, but emit many false warnings, especially when analyzing large pieces of code. On the other hand, dynamic analysis tools have much higher precision—instrumentation frameworks such as Valgrind [19], memory safety checkers such as AddressSanitizer [26] and MemorySanitizer [30], and datarace detectors such as ThreadSanitizer [27] and RoadRunner [7] have found hundreds of bugs and security vulnerabilities in large open source projects—but routinely miss bugs because of the low coverage induced by test suites. As such, while static analyses provide an upper bound on the space of program behaviors, dynamic analysis provides concrete evidence for their existence, thus establishing a lower bound. This naturally raises the question: *Can we use empirical data gathered from witnessing program executions to improve the effective accuracy of static analysis tools?*

In a parallel thread, researchers have recently developed probabilistic techniques to incorporate feedback from human users into the output of static analyzers [9, 13, 24]. These approaches build on the observation that analyzers reuse portions of their reasoning to derive multiple alarms; as a result, an inaccurate function summary, dataflow fact, or may-happen-in-parallel assertion can lead to multiple false warnings. The idea then is to recover these reasoning traces, and use them to construct a Bayesian network that captures correlations between the ground truths of each of the alarms. The user then repeatedly inspects alarms and indicates whether or not they represent real bugs. In response, the analyzer computes the conditional probabilities of the remaining alarms in light of this new information, and reprioritizes them in decreasing order of confidence. As a result, these systems are able to rapidly deprioritize false warnings, and uncover the real bugs in the program.

Despite its experimental success, Bingo [9, 24], which forms our conceptual starting point, suffers from a few important limitations: first, a purely static initialization of the Bayesian network has limited information, and requires more human guidance. This is evident in cases of *false generalization*: since the abstraction necessarily over-approximates program behaviors, a probabilistic model derived from the abstract behavior of the program sometimes causes negative feedback given by the user to erroneously propagate to the true bugs, thereby suppressing them. Furthermore, users are typically only able to answer questions about the correctness of the final warnings, and not about intermediate assertions and dataflow facts derived by the analysis, and finally, erroneous human feedback has the potential to greatly degrade the quality of ranking in subsequent iterations.

The central insight of our paper is that such interactive alarm ranking systems can incorporate feedback not just from human users, but also from diverse sources of knowledge, including by dynamic instrumentation of test cases. We begin with the warnings

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468626

and dataflow facts emitted by the Sparrow static analyzer [20, 21], which checks C programs for a range of memory safety errors, and use these to selectively instrument the program for analysis by the DFSan dynamic dataflow tracking framework [32]. We then run the instrumented program on its test inputs, and collect empirical evidence for each theoretically predicted dataflow fact. We use the resulting information as a first round of feedback to the alarm prioritization system, thus greatly improving the quality of the ranking even before human intervention. We present the architecture of this system, which we call DynaBoost, in Figure 1.

We emphasize that, in non-pathological cases, where test inputs do not themselves expose erroneous program behaviors, the feedback provided by DFSan is limited to intermediate dataflow facts rather than the final alarms raised by the static analysis. In this situation, providing feedback from dynamic analysis corresponds to providing evidence for internal nodes in the Bayesian network: the conditional independencies [22] thus created are responsible for limiting the impact of false generalization. Conversely, because of the incompleteness of static analysis, the presence of an *abstract* dataflow path from a tainted source to a sensitive sink does not, by itself, imply the possibility of tainted data causing program errors: as a result, transferring feedback from DFSan to the probabilistic ranking subsystem requires some care while engineering the Bayesian network.

We implemented these ideas using Bingo [9, 24], Sparrow [20, 21], and DFSan [32] as building blocks. We evaluated our algorithms on a suite of 13 Unix command line programs, ranging in size from 9 KLOC to 112 KLOC, and which contain a set of known historical bugs. On average, across all these benchmarks, Sparrow emits 566 warnings per program. Using our system, DynaBoost, a programmer is able to discover all these bugs after triaging just 59.5 warnings, on average per program. Notably, this is an 89% reduction compared to an unaided user, and a 35% improvement over Bingo, which requires 92.2 rounds of feedback, on average. Much of this ranking improvement is due to a dramatic reduction in the frequency and severity of false generalization events compared to Bingo (see Figure 6): on average, DynaBoost has 79% fewer false generalization events, each of which is itself only 11% of the size of the average event occurring within Bingo.

Contributions. To summarize, we make the following contributions in this paper:

(1) We develop a Bayesian framework, DynaBoost, to combine information extracted from static and dynamic program analysis.
(2) We implement a system to perform targeted dynamic instrumentation based on the results of an over-approximate static analysis.
(3) We present an experimental evaluation across a suite of Unix utilities, and demonstrate an average drop of 35% in human alarm annotation effort.

2 MOTIVATING EXAMPLE

In this section, we provide an overview of our approach by considering an example bug from the Linux command line program sort. We will discuss how DynaBoost coordinates the static analyzer, Sparrow, the dynamic analyzer, DFSan, and the Bayesian alarm prioritization process to accelerate bug discovery.

2.1 Postmortem of a Coreutils Bug

In Figure 2, we show a snippet of code adapted from Version 7.2 of the GNU coreutils program sort which contains a buffer overrun bug [4]. The program has a feature to merge a set of previously sorted files, and this functionality is implemented in the `merge` function shown in the figure. This function in turn calls another function named `avoid_trashing_input` to account for cases where one of the input files is reused as an output file.

The `avoid_trashing_input` function iterates over the input files, and checks whether they are the same as the output file. If so, then it attempts to merge the remaining files into a new temporary file on line 34. It might be unable to complete this merge due to the limited availability of file handles from the operating system, so it packs the `files` array by moving the remaining unmerged files on line 38. Unfortunately, instead of moving `nfiles - (i + num_merged)` files, the third argument requests that `num_merged` files be moved, possibly resulting in an out-of-bounds memory access.

To statically find this bug, we use the Sparrow program analyzer [21]. It uses a def-use graph computed using the sparse analysis framework [20] to determine all data dependencies leading up to sensitive memory accesses, and subsequently performs an interval analysis to prove memory safety. As one would expect, it is unable to prove the safety of the call to `memmove` on line 38, and raises an alarm at this line.

Notably, however, the underlying interval analysis is non-relational, and since the `files` array is dynamically allocated, it is also unable to show that the accesses to `files[i]` on lines 36 and 37 are safe. It therefore raises two additional alarms at these lines. Of course, these are both false warnings, and are a result of an overly coarse abstraction. However, this abstraction was deliberately chosen to allow the analysis to scale to large real-world programs, and is an example of the accuracy-scalability trade-offs routinely made by analysis designers. Overall, the sort program has 98 KLOC, and Sparrow emits 715 warnings, including the bug on line 38.

2.2 Interactive Alarm Prioritization

We will now explain the interactive alarm prioritization process used by Bingo [9, 24] that leverages a probabilistic model to generalize from user feedback to suppress likely false alarms and prioritize likely true bugs.

Reconstructing the derivation graph. The first step is to reconstruct the *reasoning trace* that causes Sparrow to report each alarm. This reasoning process—interval analysis applied to the def-use graph—can be approximately described by the derivation rules shown in Figure 3. Starting from variable definitions and one-step dataflow edges in the program, indicated by tuples of the form VarDefn(a) and DUEdge(a, b) respectively, the analyzer computes dataflow paths of the form DUPath(a, b). If the final node b in each derived DUPath(a, b) corresponds to an array access, the analyzer performs additional reasoning to prove the safety of the operation at program point b. Note that we have not modeled the details of this sub-analysis—which employs an interval abstraction—and instead provide input tuples of the form Overflow(b) as stubs for unmodeled parts of the reasoning process. When the analyzer finds such a dataflow path leading up to a potentially unsafe memory

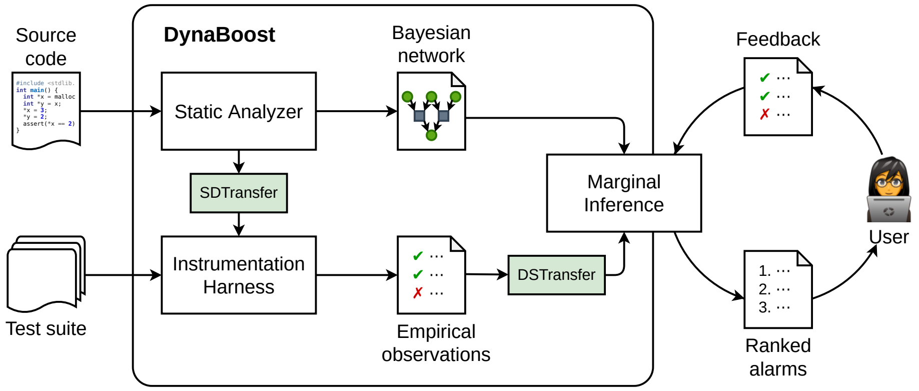


Figure 1: Architecture of the DynaBoost framework. Our main technical contributions include the construction of the Bayesian network and the boxes marked SDTransfer and DSTransfer, corresponding to the transfer of instrumentation targets from the static analyzer to the instrumentation harness, and the transfer of empirical evidence from the dynamic analysis to the probabilistic model respectively.

```
1   static unsigned int nmerge = 16;

2   struct sortfile {
3     char const *name;
4     pid_t pid;
5   };

6   void merge(struct sortfile *files, size_t nfiles, char const *output_file)
7   {
8     size_t in, out;
9     for (out = in = 0; nmerge <= nfiles - in; out++) {
10      dfsan_set_label("src9", &out, sizeof(size_t));
11      struct sortfile temp;
12      create_temp(&temp);
13      size_t num_merged = mergefiles(&files[in], nmerge, &temp);
14      in += num_merged;
15      files[out].name = temp.name;
16      files[out].pid = temp.pid;
17    }

18    memmove(&files[out], &files[in], (nfiles - in) * sizeof(*files));
19    nfiles -= in - out;
20    nfiles = avoid_trashing_input(files, nfiles, output_file);
21    ···
22  }
```

```
23  size_t avoid_trashing_input(struct sortfile *files, size_t nfiles, char const *outfile)
24  {
25    print(dfsan_get_label(nfiles));

26    for (size_t i = 0; i < nfiles; i++) {
27      bool same = ···;
28      if (same) {
29        size_t num_merged = 0;
30        while (i + num_merged < nfiles) {
31          print(dfsan_get_label(i));
32          struct sortfile temp;
33          create_temp(&temp);
34          num_merged += mergefiles(&files[i], nfiles - i, &temp);
35          print(dfsan_get_label(num_merged));
36          files[i].name = temp.name;
37          files[i].pid = temp.pid;

38          memmove(&files[i + 1], &files[i + num_merged], num_merged * sizeof(*files));
39          nfiles -= num_merged - 1;
40        }
41      }
42    }

43    return nfiles;
44  }
```

Figure 2: Code fragment adapted from the sort program. The lines highlighted in red correspond to the alarms raised by the Sparrow static analyzer, while the lines highlighted in green correspond to the instrumentation added by DynaBoost.

access, it reports an alarm at the appropriate point, as indicated by the derivation rule r_3.

In the case of our example, the assignment to the variable out on line 9 may influence the value of nfiles at line 19 and through the call to avoid_trashing_input, affect the values of variables i and num_merged at lines 30 and 34 respectively. The program subsequently accesses the i-th and (i + num_merged)-th elements of the files array at the locations of the alarms raised by Sparrow. We may visualize this reasoning trace as the derivation graph shown in Figure 4.

A probabilistic model of alarms. Observe now that if the user triages the alarm at line 36 and indicates that it is *not* a real bug, then we may conclude that line 37 is also a false alarm. The prioritization algorithm infers such correlations between alarms using the derivation graph in Figure 4.

As an example, consider the tuple DUPath(9, 38), which indicates that data may flow from a source at location 9 to a variable at location 38. Because of inaccuracies in the construction of the dataflow graph, there is a small, but non-zero probability that a

Input relations

$\mathrm{VarDefn}(a)$:	Variable definition at program point a
$\mathrm{Overflow}(b)$:	Possible buffer overflow at point b
$\mathrm{DUEdge}(a, b)$:	Direct dataflow edge between program points a and b

Output relations

$\mathrm{DUPath}(a, b)$:	(Transitive) Dataflow path from program point a to b
$\mathrm{Alarm}(b)$:	Alarm indicating possible buffer overflow at b

Derivation rules

$$\begin{aligned} r_1 &: \mathrm{DUPath}(a, b) \coloneq \mathrm{VarDefn}(a), \mathrm{DUEdge}(a, b) \\ r_2 &: \mathrm{DUPath}(a, c) \coloneq \mathrm{DUPath}(a, b), \mathrm{DUEdge}(b, c) \\ r_3 &: \quad \mathrm{Alarm}(b) \coloneq \mathrm{DUPath}(a, b), \mathrm{Overflow}(b) \end{aligned}$$

Figure 3: Modeling the program analyzer using derivation rules, represented here as a Datalog program.

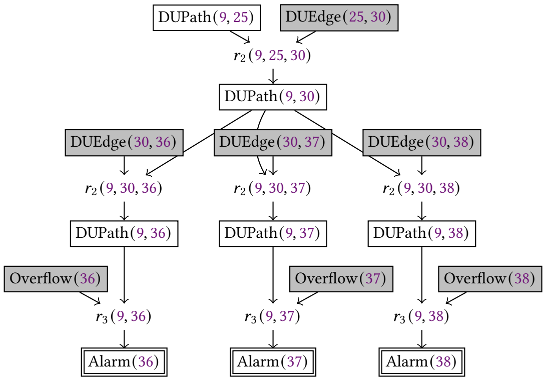


Figure 4: Derivation graph for the alarms in Figure 2. Each clause node (e.g., $r_2(9, 30, 38)$) indicates the variable valuation with which the corresponding rule in Figure 3 was fired.

given derived tuple of this form does not represent a viable dataflow path in the program.

To model such inaccuracies, the alarm prioritization process interprets the derivation graph as a Bayesian network, and associates each of its clauses with a probability of "*misfiring*",

$$\Pr(\neg t \mid t_1 \wedge t_2 \wedge \cdots \wedge t_k),$$

where t is the tuple produced by instantiating a rule r with the appropriate input tuples, $t_1, t_2, \ldots, t_k$. From this, one can compute the probability of each alarm, Alarm(c), being a real bug, Pr(Alarm(c)), and use these probabilities to order alarms for triaging, with high-confidence alarms inspected before alarms with low confidence. Furthermore, the computation of marginal probabilities—$\Pr(\text{Alarm}(c_1) \mid \text{Alarm}(c_2) \wedge \neg\text{Alarm}(c_3))$, for example—provide a natural mechanism by which to generalize from user feedback and suppress or prioritize similar warnings from the static analyzer.

For simple networks, and for the sake of exposition, we may compute these values by hand, as we demonstrate in Appendix A. If we assume that the prior probability Pr(DUPath(9, 25)) = 0.9, and that there is a 1% i.i.d probability of each rule misfiring, then it follows that the prior probability of Alarm(37) is given by:

$$\Pr(\text{Alarm}(37)) = 0.873. \tag{1}$$

Generalizing from user feedback. If the user indicates that Alarm(36) is a false warning, we would be interested in the values of $\Pr(\text{Alarm}(37) \mid \neg\text{Alarm}(36))$ and $\Pr(\text{Alarm}(38) \mid \neg\text{Alarm}(36))$:

$$\Pr(\text{Alarm}(37) \mid \neg\text{Alarm}(36)) = 0.137. \tag{2}$$

Observe the dramatic drop in $\Pr(\text{Alarm}(37) \mid \neg\text{Alarm}(36))$ compared to the value of Pr(Alarm(37)), which occurs because of the shared derivation graph between Alarm(36) and Alarm(37). As a result, the user feedback causes us to suppress the second alarm, and permit the accelerated discovery of bugs in other parts of the codebase. Recall that Sparrow reports 715 alarms when it analyzes GNU sort; the user-in-the-loop interaction process we just discussed causes the bug to be discovered after inspecting only 176 alarms. We graphically depict this interaction loop in the rightmost portion of the system diagram in Figure 1, and will review the construction of the probabilistic model in Section 3.

Despite the significant empirical improvement provided by Bingo, it suffers from several important limitations. In principle, the conditional probabilities, $\Pr(\text{Alarm}(c) \mid t)$, may be computed with respect to *any* tuple t produced by the analysis, and not just those corresponding to alarms, $t = \text{Alarm}(c')$. In practice, however, because of the highly technical nature of the analysis, users are only able to provide limited forms of feedback, and even then, are prone to making mistakes. Second, an investigation of the interaction process reveals numerous instances of *false generalization*, where the rank of the real bug drops during a single iteration. See, for example, the "spikes" in the plots of Figure 6, and the aggregate statistics in Table 3. Finally, because the triage process is primarily user-driven, it sometimes results in a lengthy interaction process in which the user has to inspect a large number of false warnings before discovering real bugs in the program. These limitations provide the background for our present paper.

2.3 Dynamic Instrumentation as an Information Source

Our central insight. The central insight of our paper is that the feedback provided to the alarm prioritization algorithm need not only come from human users, but can be drawn more broadly from any source of information about the program. In particular, we demonstrate the possibility of using dynamic information obtained from test executions as an information source.

The dynamic dataflow sanitizer DFSan. DFSan [32] provides two functions to inject taint labels and inspect the taint values of variables at runtime: (a) `dfsan_set_label(dfsan_label label, void *addr, size_t size)`, which associates the sequence of memory locations `addr`, `addr + 1`, `addr + 2`, ..., `addr + (size - 1)` with the taint value `label`, and (b) `dfsan_get_label(long data)`, which returns the taint label associated with the value `data`. The annotated program is then instrumented by DFSan during compilation so as to track these injected taint values as the program executes.

Dynamic instrumentation. For each dataflow path, DUPath(a, b), reported by Sparrow, we use DFSan to monitor program executions for concrete evidence of a flow between the (a, b) source-sink pair. We highlight a simplified version of the annotations applied to the sort program in green in Figure 2. As an example, consider the tuple DUPath(9, 25). We annotate the variable being assigned at the source, `out`, with a distinguished taint value—here `"src9"`—with the call to `dfsan_set_label` on line 10. We then retrieve the labels of all predicted downstream sinks with calls to `dfsan_get_label` on lines 25, 31, and 35, and check whether the source taint propagates to each of the sink locations. We run this instrumented program on all tests provided with GNU sort Version 7.2, looking for experimental confirmation of the predicted source–sink flows. This in turn enables us to provide early feedback to the Bayesian ranking process, so that it now ranks alarms according to $\Pr(\text{Alarm}(c) \mid \text{DUPath}(a, b))$, rather than merely by their prior probabilities Pr(Alarm(c)).

To motivate the value of this process, we will now discuss how Bingo causes false generalization while analyzing sort. First, consider the symmetry between Alarm(36), Alarm(37), and Alarm(38) in Figure 4. From this with our previous calculation of the prior and posterior probabilities in Equations 1 and 2, we conclude that:

$$\Pr(\text{Alarm}(38)) = 0.873, \quad \Pr(\text{Alarm}(38) \mid \neg\,\text{Alarm}(36)) = 0.137.$$

This drop in posterior probability causes the alarm to drop in the ranking, compared to other warnings in the program, and, since this represents a real bug, corresponds to a false generalization event. Overall, over the course of the interaction process, Alarm(38) gets deprioritized twice, corresponding to the user inspecting each of the neighboring alarms, Alarm(36) and Alarm(37). Visually, these correspond to the two prominent spikes in Figure 6f.

We will now describe how DynaBoost mitigates this problem of false generalization. Observe that even though the functionality implemented in `avoid_trashing_input` is not exercised by any test input, the function is always called from `merge`, and the standard suite of test cases does attempt to merge files. As a result, DFSan observes experimental evidence for DUPath(9, 25). The algorithm then ranks alarms in terms of $\Pr(\text{Alarm}(c) \mid d \wedge e)$, where $d = \text{DUPath}(9, 25)$ is the dynamic feedback, and e represents the feedback provided by the user. In this case, the original probabilities are given by:

$$\begin{aligned}\Pr(\text{Alarm}(36) \mid d) &= \Pr(\text{Alarm}(37) \mid d) \\ &= \Pr(\text{Alarm}(38) \mid d) = 0.99^3 = 0.970, \qquad (3)\end{aligned}$$

where, as before, the expression 0.99^3 arises because of the three rule applications between the evidence and the query nodes. After the first round of user feedback, the posterior probability of Alarm(38) is given by $\Pr(\text{Alarm}(38) \mid d \wedge e)$, where $e = \neg\,\text{Alarm}(36)$. We sketch this calculation in Appendix A, from which we can conclude that: $\Pr(\text{Alarm}(38) \mid d \wedge \neg\,\text{Alarm}(36)) = 0.650$. Observe that user feedback on the false alarm, Alarm(36), causes a much smaller drop in confidence in Alarm(38), and a significantly smaller false generalization event.

Experimental results. As such, DynaBoost addresses the previously outlined limitations in Bingo: First, dynamic feedback, provided to tuples of the form DUPath(a, b), represents program behaviors which users would find difficult to certify. Second, this additional information constrains the ways in which the marginal inference algorithm may propagate user feedback—for example, by activating conditional independencies in the Bayesian network—and this reduces the incidence of false generalization. For example, compare the frequency and magnitude of spikes of DynaBoost and Bingo in the plots of Figure 6: there are 79% fewer spikes, and each spike is only 11% of the original size. As a consequence, dynamic feedback becomes a valuable auxiliary source of information, and dramatically reduces the alarm inspection burden, by approximately 35% compared to Bingo, and approximately 89% compared to an unaided user (See Table 2).

3 AN OVERVIEW OF BAYESIAN ALARM PRIORITIZATION

We will now present a high-level description of the Bayesian alarm prioritization framework. It conceptually works in three phases: by extracting the derivation graph from the static analyzer, converting this derivation graph into a Bayesian network, and finally engaging in an interaction loop with the user, while repeatedly performing marginal inference to rank alarms.

First, we model the static analysis as a Datalog program, such as that shown in Figure 3. Most briefly, a Datalog program consists of a set of universally quantified Horn clauses, which we call *rules*. We may read the rules from right-to-left, while treating the ":–" operator in the middle read as the implication operator, " $\Longleftarrow$ ". For example, the rule r_2 may be read as, "For all program points a, b, c, if there is a dataflow from a to b, DUPath(a, b) and a one-step flow from b to c, DUEdge(b, c), then, transitively, there is also a dataflow from a to c, DUPath(a, c)." We will refer to each input hypothesis and output conclusion, of the form $R(c_1, c_2, \ldots, c_n)$, as a *tuple*.

We then modify the static analyzer to provide explanations for each of its alarms. These explanations take the form of a derivation graph, such as that shown in Figure 4. The derivation graph G is a (possibly cyclic) directed graph consisting of two types of nodes, corresponding to the tuples and grounded clauses of the least fixpoint of the Datalog program. Each clause node refers to a specific instantiation of a rule, which takes several tuples as hypotheses and produces a tuple as conclusion. In our diagrams, we will indicate the tuples by boxed vertices, and leave the clause nodes unboxed.

We remark that the derivation graph is a best-effort post hoc explanation: Sparrow is itself written in unrestricted OCaml, and there are portions of the analyzer—such as the interval analysis—which are left unmodeled. It should be possible to extract similar derivation graphs from other static analyzers. To obtain this derivation graph, we reused the modifications to Sparrow which were originally employed in Drake, and which consists of approximately 500 lines of changes to a 15 KLOC codebase [9].

Next, we convert the derivation graph into a Bayesian network. We associate each node of the graph with a conditional probability distribution, which indicates the probability of the node being true for each combination of truth values of its hypothesis nodes.

Consider a grounded clause $g = r(c_1, c_2, \ldots, c_k)$ which applies rule r to produce the conclusion t from hypotheses $t_1, t_2, \ldots, t_n$: $t \overset{r}{\Longleftarrow} t_1 \wedge t_2 \wedge \cdots \wedge t_n$. The clause nodes may be thought of as conjunctions, which fire only when all of its hypothesis nodes are derivable. To model the approximations of the static analysis, we allow for the possibility of clause nodes misfiring, with a small rule-dependent misfiring probability, $1 - p_r$:

$$\Pr(r(c_1, c_2, \ldots, c_k) \mid t_1, \ldots, t_n) = \begin{cases} 1 - p_r & \text{if } t_1 \wedge \cdots \wedge t_n, \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

So that the probabilities all add up to 1, we naturally have $\Pr(\neg g \mid t_1, \ldots, t_n) = 1 - \Pr(g \mid t_1, \ldots, t_n)$. While these firing probabilities p_r may be determined using techniques such as expectation maximization, we follow Bingo and uniformly set them to 0.99.

Similarly, the tuple nodes of a derivation graph may be thought of as disjunctions, which are derivable only when at least one of its contributing clauses is able to fire. Consider a tuple t which is the result of several alternative clauses: $g_1, g_2, \ldots, g_k$. We model t as a deterministic disjunction:

$$\Pr(t \mid g_1, \ldots, g_k) = \begin{cases} 1 & \text{if } g_1 \vee \cdots \vee g_n, \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

As before, to indicate exhaustiveness, $\Pr(\neg t \mid g_1, \ldots, g_k) = 1 - \Pr(t \mid g_1, \ldots, g_k)$. For all input tuples t_{in}, we define the prior probability as 1: $\Pr(t_{\text{in}}) = 1$.

One notable challenge in the construction of this probabilistic model is that Bayesian networks are required, by definition, to be acyclic, while the derivation graph may potentially have cycles. To address this, Bingo applies a cycle elimination algorithm which drops clauses so as to obtain an acyclic graph while still preserving the derivations of all alarms. This results in an acyclic derivation graph, which is then used to build the Bayesian network.

Given the Bayesian network, we use an off-the-shelf solver, libDAI, to perform marginal inference and rank alarms for user inspection [15].

4 TARGETED INSTRUMENTATION AND FEEDBACK TRANSFER

In this section, we explain the operation of the DynaBoost system. We formally present the top-level procedure in Algorithm 1. The main contributions of this paper are in the dynamic analysis performed in Steps 3–6. We describe the core SDTransfer and DSTransfer procedures in Algorithms 2 and 3 respectively.

Algorithm 1 DynaBoost($\mathcal{A}, P, \mathcal{T}$), where $\mathcal{A}$ is the static analysis, P is the program, $\mathcal{T}$ is the set of available test cases.

```
(1) Let $S = \mathcal{A}(P)$. Statically analyze the program. The result $S$
    consists of the set of alarms, the intermediate conclusions,
    and the derivation graph connecting them.
(2) Construct the Bayesian network $B = \text{MakeBNet}(P, S)$.
(3) Let $P' = \text{SDTransfer}(P, S)$. Instrument the program using
    the static analysis output.
(4) Compute $D = P'(\mathcal{T})$. Run the instrumented program on the
    test inputs.
(5) Let $F_{Dyn} = \text{DSTransfer}(D, S)$. Determine which dataflow
    facts can be dynamically observed.
(6) Initialize the feedback, $F := F_{Dyn}$. Assert $F_{Dyn} \subseteq \text{Tuples}(S)$.
(7) Initialize the set of unlabelled alarms, $A_u := \text{Alarms}(S)$.
(8) While $A_u \neq \emptyset$:
    (i) Present the highest probability unlabelled alarm for user
        inspection:
            $a_t = \arg\max_{a \in A_u} \Pr(a \mid F).$
    (ii) If the user marks $a_t$ as true, update $F := F \wedge a_t$. Otherwise
         update $F := F \wedge \neg a_t$.
    (iii) Update $A_u := A_u \setminus \{a_t\}$.
```

4.1 Performing Targeted Instrumentation

We will now describe SDTransfer, the process of instrumenting the program based on results from the static analyzer. We present the overall algorithm in Algorithm 2.

As discussed in Section 2, in addition to the runtime instrumentation applied to the LLVM IR, DFSan provides two functions: dfsan_set_label to insert taint values into memory locations, and dfsan_get_label to retrieve the taint values associated with a value. For each dataflow tuple DUPath(a, b) produced by the static analyzer, our goal is to insert the appropriate taint values into variables being assigned at program location a using dfsan_set_label, and to retrieve the taint values from variables being used at program location b using dfsan_get_label.

The main challenge in this process is in translating program locations from Sparrow's representation to points in the program source code. Since Sparrow works with an SSA-form of the program, some program locations such as ϕ-nodes cannot be mapped back to locations in the original program. In such cases, we do not instrument the resulting DUPath(a, b) tuple. Furthermore, operations may have side-effects (such as `*p++ = 1`), syntactic constructs may be arbitrarily nested (such as `x = y->b + c`), and a single line of code may have multiple assignments (this commonly arises in `for`-loops) so that we are only able to perform a best-effort instrumentation of the source code, and omit tuples which cannot be instrumented. Overall, in our experiments in Section 5, we have a 58% success rate in instrumenting 43465 target locations.

Algorithm 2 SDTransfer(P, S). Given a program P and statically determined dataflow facts S, produces a program P' with runtime instrumentation enabled.

```
(1) For each predicted dataflow tuple DUPath$(a, b) \in S$, if $a$ and
    $b$ can both be mapped to source locations, add the instru-
    mentation highlighted in green below:

          x = ...; // Program point a
        + dfsan_set_label("src-a", &x, sizeof(x));
          ...
        + print(dfsan_get_label(y));
          read(y); // Program point b

(2) Return the instrumented program $P'$.
```

4.2 Transferring Runtime Output to Static Feedback

We run the instrumented program P' on the provided test cases, $\mathcal{T}$, and collect the list of all dataflow paths which are empirically observed. We then perform lightweight feedback enhancement to recover information about uninstrumented dataflows to or from empty nodes, and use the frequency of observation of each dataflow path to provide weighted feedback to the marginal inference algorithm. We outline this process in Algorithm 3.

By borrowing terminology from graph theory, we term a tuple DUPath(a, c) as an (a, b)-*bridge* if (a) both DUPath(a, b), DUPath$(a, c) \in S$, the predictions of the static analyzer, and (b) the clause

$$\text{DUPath}(a, b) \Longleftarrow^{r_2} \text{DUPath}(a, c) \wedge \text{DUEdge}(c, b)$$

is the only way to derive DUPath(a, b), where the rule r_2 is drawn from Figure 3. Bridges provide indirect evidence of dataflows to empty nodes c, which cannot themselves be directly instrumented. In these cases, we can use dynamic observations of DUPath(a, b) to infer truth of the bridge DUPath(a, c). In Step 1 of Algorithm 3, we repeatedly perform this feedback enhancement.

Additionally, to account for the frequency of observations of individual dataflows, we count the number of test cases #(a, b)

which witness each dataflow DUPath(a, b), and compare it to the total number of test inputs, n. This allows us to prioritize feedback to commonly observed dataflow paths, and only provide weak experimental feedback for less frequently observed dataflow paths.

Algorithm 3 DSTransfer(D, S). Given the results of the static analysis S, and the list of empirically observed dataflow facts D, computes the dynamic feedback F_{Dyn}.

1. Perform feedback enhancement. For each tuple DUPath$(a, b) \in D$, if there is an (a, b)-bridge, DUPath$(a, c) \notin D$, then update:
   $$D \coloneqq D \cup \{\text{DUPath}(a, c)\}.$$
   Repeat until fixpoint.
2. Construct the dynamic feedback F_{Dyn}.
   (i) Initialize $F_{Dyn} \coloneqq \emptyset$.
   (ii) For each tuple DUPath$(a, b) \in D$, let $\#(a, b)$ be the number of test inputs which trigger DUPath(a, b), and let n be the total number of test inputs. Provide feedback to the (a, b) dataflow by updating:
   $$F_{Dyn} \coloneqq F_{Dyn} \cup \{\text{DUPath}(a, b) \mapsto \#(a, b)/n\}. \tag{6}$$
3. Return F_{Dyn}.

4.3 Differentiating Unfiltered Dataflows

In the last step, we modify the derivation graph to differentiate between filtered and unfiltered dataflows. Because DFSan only provides an under-approximation of feasible behaviors, it does not provide information about the operations applied to the data along the (a, b) dataflow path. As a result, it is insufficient to conclude that this path could form the basis of a buffer overflow. To describe the limited information coming from DFSan, we introduce a new output relation TDUPath, which indicates the possibility of an unfiltered dataflow from source a to sink b. We use the base DUPath(a, b) tuples to derive tuples of the form TDUPath(a, b), but only apply the feedback in F_{Dyn} to the original DUPath tuples. We describe these modified Datalog rules in Figure 5. As we demonstrate in Table 2, modeling these correlations is crucial to the experimental effectiveness of DynaBoost, as it reduces the average number of iterations from 156 (for Bingo$_{all}$) to 60 (for DynaBoost$_{all}$).

5 EXPERIMENTAL EVALUATION

To evaluate the experimental effectiveness of DynaBoost, we focus on the following research questions:

RQ1. Does DynaBoost effectively prioritize the real bugs, and how does it compare to Bingo?

RQ2. Does DynaBoost reduce the frequency and magnitude of false generalization events?

RQ3. How does the number of test cases affect the ranking quality?

RQ4. How is the modification for the network structure important for utilizing the runtime feedback?

We begin this section by describing our experimental setting, and we focus on each of the above questions in Sections 5.2–5.4.[1]

Input relations: VarDefn(a), Overflow(b), DUEdge(a, b)

Output relations: DUPath(a, b), Alarm(a, b)
TDUPath(a, b): (Unfiltered) Dataflow path from program point a to b

Derivation rules

r_1 : $\text{DUPath}(a, b) \mathrel{:-} \text{VarDefn}(a), \text{DUEdge}(a, b)$
r_2 : $\text{DUPath}(a, c) \mathrel{:-} \text{DUPath}(a, b), \text{DUEdge}(b, c)$
r_3' : $\text{Alarm}(b) \mathrel{:-} \text{TDUPath}(a, b), \text{Overflow}(b)$
r_4 : $\text{TDUPath}(a, b) \mathrel{:-} \text{DUPath}(a, b), \text{VarDefn}(a), \text{DUEdge}(a, b)$
r_5 : $\text{TDUPath}(a, c) \mathrel{:-} \text{DUPath}(a, c), \text{TDUPath}(a, b), \text{DUEdge}(b, c)$

Figure 5: Modified derivation rules to capture unfiltered dataflows. We reuse rules r_1 and r_2 from Figure 3, and replace rule r_3 with r_3'.

Table 1: Benchmark characteristics. Size and #Test report the lines of code and the number of test cases.

Program	Version	Analysis	Size(KLOC)	Tests
bc	1.06	Interval	14	18
cflow	1.5	Interval	40	33
grep	2.19	Interval	68	1646
gzip	1.2.4a	Interval	9	49
libtasn1	4.3	Interval	30	17
patch	2.7.1	Interval	51	189
readelf	2.24	Interval	65	2601
sed	4.3	Interval	83	522
sort	7.2	Interval	98	789
tar	1.28	Interval	112	699
optipng	0.5.3	Taint	61	176
latex2rtf	2.1.1	Taint	27	130
shntool	3.0.5	Taint	13	7

5.1 Experimental Setup

Choice of benchmarks. We ran DynaBoost on a suite of widely used C programs shown in Table 1. All benchmarks are from previous work using Sparrow [8, 9] and recent CVE reports. We excluded benchmarks with less than 5 KLOC because the limited number of alarms raised by Sparrow does not impose a significant alarm inspection burden. We additionally excluded wget and urjtag because of compatibility issues either with DFSan or with Clang.

We used the test inputs that come with the program, if available, to collect dynamic information. Three benchmark programs did not have a developer-provided test suite: gzip, shntool, and optipng. For these programs, we collected sample audio and image files [5, 33] (shntool, optipng), and compressed files from the Canterbury corpus [23] (gzip).

Dynamic instrumentation and ranking process. We use Dataflow-Sanitizer (DFSan) [32] to collect the runtime dataflow information. For each source–sink pair, DUPath(a, b) reported by Sparrow, we determine the variables assigned at program location a and inject a runtime taint label using the `dfsan_set_label()` function, and we retrieve the taint labels of all variables accessed at program location b using the `dfsan_get_label()` function. We break ties between

[1] We will make our benchmarks and implementation public upon paper acceptance.

identically ranked alarms by using their confidence values from the purely static ranking process in Bingo.

Baselines. We instantiate DynaBoost with two different settings: DynaBoost$_{all}$ that is based on the augmented network structure of Section 4.3 and initialized with dynamic feedback, and DynaBoost$_{zero}$ which also uses the new network structure, but withholds dynamic feedback. We similarly instantiate two baselines: Bingo$_{all}$ which uses the original network structure [24] but also includes dynamic feedback, and Bingo$_{zero}$ which neither uses the new network structure nor uses dynamic feedback [24].

Runtime performance of DynaBoost. Sparrow requires an average of 206 seconds to analyze the benchmark programs. This ranges from a few seconds for gzip to 840 seconds for tar. Instrumentation and dynamic data collection requires a comparable amount of time, ranging from a few seconds to about 20 minutes. Note that this is the time for all test inputs, run in sequence, and can be parallelized in a straightforward manner. Finally, the initial ranking and reprioritization processes are much faster, and Step 8(i) of Algorithm 1 takes 14 seconds, on average.

5.2 RQ1: Effectiveness of Ranking

We first evaluate the effectiveness of DynaBoost$_{all}$ compared to Bingo$_{zero}$. Notice that DynaBoost$_{all}$ is different from Bingo$_{zero}$ in two ways: feedback from dynamic analysis and augmented Bayesian network structure. These aspects will be further discussed in the subsequent sections. In this section, we measure the initial rankings of all bugs and the number of iterations until DynaBoost$_{all}$ and Bingo$_{zero}$ find all the bugs. The results are shown in Table 2.

We observe that DynaBoost$_{all}$ is significantly more effective at prioritizing true bugs compared to Bingo$_{zero}$. On average, the full system, DynaBoost$_{all}$ finds the bug within 59.5 iterations while Bingo$_{zero}$ requires 92.2 iterations, resulting in a 35% reduction in the human alarm inspection burden. One of the main reasons of this effectiveness is the improvement of the quality of initial rankings (i.e., right after transferring feedback from DFSan). Before any user feedback, DynaBoost$_{all}$ places the true bugs at rank 160 while Bingo$_{zero}$ yields 251, on average. This statistic also would be of interest in cases where users do not wish to interact with the tool, but merely want a listing of alarms above a certain threshold for offline inspection. Another reason is the reduction of false generalization which will be discussed more in the next section.

We notice particularly dramatic improvements in the cases of cflow and tar. Both of the benchmarks show the improved quality of the initial rankings (from 517 to 173 for cflow, from 697 to 253 for tar), thereby producing the reduced number of iterations needed to find all the bugs (78% and 58% improvements for cflow and tar respectively). Interestingly, while both of these benchmarks already had test cases that exercise the data flow paths in question, they did not capture the bugs because they could only be triggered by carefully chosen test inputs. As a result, the only remaining uncertainty in the ground truth of the alarm arises from incompleteness in the interval analysis performed by Sparrow. Therefore, DynaBoost$_{all}$ quickly prioritizes these alarms over the rest.

Finally, we notice small performance regressions on some benchmarks. For the buffer overflow benchmarks such as readelf, these were caused by a biased test suite which fails to exercise any of the dataflow facts that are involved in the derivation of the bugs. In the case of shntool, we speculate that the regression is the result of fluctuations caused by the small number of alarms emitted by Sparrow(23) combined with the large number of true alarms (6), which together amplifies the effect of noise in the ranking process. In any case, we note that for all these benchmarks, the absolute value of the performance regression is small (≤ 13 iterations), and the overall ranking process still provides massive reductions in the human alarm inspection burden.

5.3 RQ2: Reduction of False Generalization

Next, we measure the impact of dynamic feedback in reducing false generalization. After each round of feedback, we measure the average rank of all real bugs, and compare this average to their average rank in the previous round. We define a *false generalization event* as one in which this average drops by 10% or more and by at least 5 alarms. The rank reported in Table 3 is the sum of this average rank drop across all false generalization events, and shows how DynaBoost$_{all}$ mitigates the false generalization problem.

In case of sort, DynaBoost$_{all}$ reduces the number of required user iterations by 39.7%. According to the results, the average number of rank drop events of true alarms is reduced from 4.5 to 0.9. The average rank drop size for each time is also dropped from 214.8 to 23.1. While Bingo$_{zero}$ introduces two major false generalizations around iteration 100 and 130, DynaBoost$_{all}$ substantially reduces their impact on the two same false alarms around iteration 75. As shown in Figure 4, the derivation rules of all these alarms share the tuple observed at runtime, DUPath(9, 25). Thus, for DynaBoost$_{all}$, when rejecting the two false alarms, we do not decrease the associated probabilities blindly, instead, we limit the extent of feedback propagation because of the confidence brought by the observation.

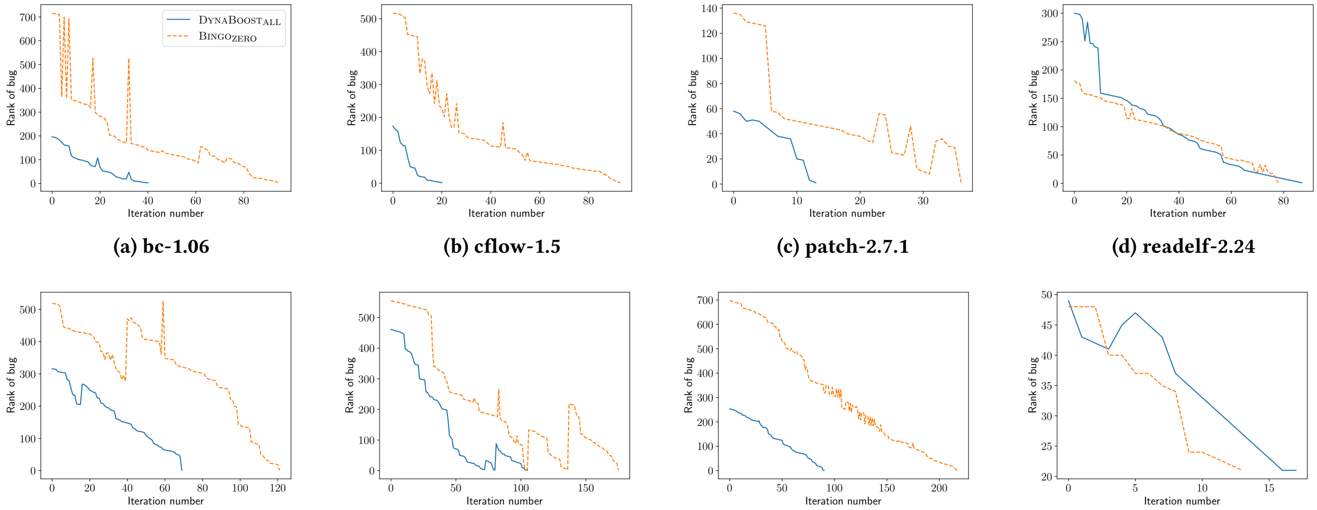
5.4 RQ3: Impact of Test Cases on Ranking Performance

In this section, we conduct a sensitivity study with different amounts of test data. Since DynaBoost leverages dynamic analysis results, the quality of ranking relies on the number and coverage of test cases. To quantify this relationship, we ran DynaBoost on the benchmarks with different subsets of the entire test suite, and measured the number of iterations needed to discover all bugs in the programs. We varied the fraction of test cases chosen, and repeated the experiment for each fraction 10 times.

We plot these results in Figure 7. We additionally include the number of iterations needed by Bingo$_{zero}$ as a visual baseline (dotted lines). Notice that the left-most observation of each benchmark, corresponding to column DynaBoost$_{zero}$ in Table 2, already corresponds to 72% reduction in alarm inspection burden compared to an unaided user, averaged across all benchmarks. We observe that for most of the benchmarks, even when only half of the total test inputs are chosen, the number of iterations needed by DynaBoost remains close to its effectiveness on the complete test suite. In fact, for a majority of benchmarks, this is true even when we provide

Table 2: Effectiveness of DynaBoost compared to Bingo. #Alarms shows the total number of alarms reported by Sparrow. Init and Iters measure the average initial rank of the bugs and the number of iterations needed until finding all the bugs.

Program	#Alarms	Bugs	DynaBoost$_{all}$		DynaBoost$_{zero}$		Bingo$_{all}$		Bingo$_{zero}$	
			Init	Iters	Init	Iters	Init	Iters	Init	Iters
bc	535	2	121.0	48	327.0	113	84.0	100	358.0	96
cflow	805	1	173.0	21	356.0	105	163.0	163	517.0	94
grep	912	1	144.0	66	714.0	378	44.0	44	72.0	53
gzip	344	14	230.0	235	229.5	326	233.4	287	147.3	283
libtasn1	357	1	5.0	14	113.0	6	3.0	5	33.0	9
patch	502	1	58.0	14	227.0	33	27.0	27	136.0	36
readelf	882	1	300.0	88	111.0	36	460.0	443	181.0	78
sed	819	1	316.0	70	469.0	196	284.0	284	519.0	122
sort	715	1	461.0	106	479.0	174	458.0	446	555.0	176
tar	1369	1	253.0	91	602.0	220	162.0	162	697.0	218
optipng	67	1	3.0	4	5.5	4	11.0	41	7.0	6
latex2rtf	13	2	6.5	5	2.0	2	6.5	6	30.0	14
shntool	23	6	8.2	18	15.2	21	7.7	18	8.0	13
Average	564.8	2.5	159.9	60	280.8	124	149.5	156	250.8	92



(a) bc-1.06

(b) cflow-1.5

(c) patch-2.7.1

(d) readelf-2.24

(e) sed-4.3

(f) sort-7.2

(g) tar-1.28

(h) shntool-3.0.5

Figure 6: Ranking changes of true alarms by DynaBoost$_{all}$ and Bingo$_{zero}$. The remaining plots are available in Appendix B.

just 10% of all test inputs. Furthermore, the variation in number of iterations quickly disappears as test cases are added.

One striking observation in Figure 7 is that for many benchmarks, the number of iterations needed by DynaBoost is independent of the size and choice of test inputs. We conjecture that many test inputs exercise similar paths through the program, such as by entering through `main()`, parsing command line arguments, or by exercising common functionality, such as parsing regular expressions in grep. Such inputs would result in many shared dataflows, which are responsible for similar prioritization results. On the other hand, some programs have a few distinct functionalities, such as tar which can alternately compress or decompress a file, and sampling test inputs leads to a bimodal performance distribution, as we can see in Figure 7f.

We observe notably exceptional behavior for readelf as the number of iterations degrades with additional test cases. According to our investigations, this is because no test case ever explores the buggy function, `process_cu_tu_index()`. This function is responsible for reading the contents of `dwo` files, which contain DWARF objects related to debug information in the binary. We subsequently chose an intermediate tuple in the derivation tree and manually provided positive feedback, thus overriding data obtained from the test cases. This reduced the number of iterations needed to find the bug from 88 to 49. We conclude that the biased set of test cases

Table 3: Magnitude and frequency of false generalization. #Events indicates the number of false generalization events, and Rank↓ indicates the sum of the average rank drop across all false generalization events.

Program	DynaBoost$_{all}$		Bingo$_{zero}$	
	Rank ↓	# Events	Rank ↓	# Events
bc	32.5	2	661.0	6
cflow	0	0	421.0	7
grep	0	0	0	0
gzip	55.6	6	225.9	14
libtasn1	0	0	0	0
patch	0	0	72.0	3
readelf	33.0	1	43.0	3
sed	63.0	1	360.0	2
sort	116.0	2	459.0	4
tar	0	0	485.0	16
optipng	0	0	65.0	3
latex2rtf	0	0	0	0
shntool	0	0	0	0
Average	23.1	0.9	214.8	4.5

in readelf continuously prioritizes alarms in other functions and suppresses the true bug.

5.5 RQ4: Impact of Network Structure on Ranking Performance

Finally, we empirically clarify the impact of augmented network structure described in Section 4.3. We compare the performance of DynaBoost$_{all}$ compared to Bingo$_{all}$ that is based on the original network with full feedback from dynamic execution.

The results are shown in Table 2. With the original network, the number of required iterations by Bingo$_{all}$ is 2.6x higher than that for DynaBoost$_{all}$. For example, we observed significant regressions for readelf, sed and sort with Bingo$_{all}$.

Interestingly, the performance of Bingo$_{all}$ is even worse than Bingo$_{zero}$. Even though dynamic feedback improves the quality of the initial ranking by 40% in the original network, it appears that the newly introduced conditional independencies by the feedback heavily limit positive generalization of user feedback. For example, Bingo$_{all}$ did not generalize any feedback for benchmarks (e.g., cflow , grep, patch, sed, and tar), but just enumerate alarms following the initial rankings. This result shows that the new network structure is more suitable for handling dynamic feedback.

6 LIMITATIONS AND THREATS TO VALIDITY

One significant restriction of our experimental evaluation is that we have restricted our attention to a single static analyzer (Sparrow), two analyses (buffer overflows and taint tracking), and a small set of benchmark programs. However, our principal assumptions are that the analysis permit recovery of the derivation graph (such as Figure 4), and that it permit experimental observation of intermediate facts.

For example, def-use chains are a general building block for a large class of static analysis tools based on the sparse analysis framework [20], including TAJS [10], Pinpoint [28], and SVF [31]. Bug-finding tools based on these techniques can directly leverage our work as described in the paper.

Furthermore, if the analysis is expressed in Datalog or using similar deductive approaches—examples include Chord [16] and Doop [2]—and if the abstract behaviors are experimentally observable, then our techniques are again potentially applicable. As an example, the datarace detector in Chord fundamentally depends on a may-happen-in-parallel analysis, which can be experimentally observed using dynamic datarace detectors such as RoadRunner [7].

Another threat to the validity of our experiments arises from our protocol. We started with a set of historical bugs for each of the programs, and assumed that only this explicitly identified target bug was real, and that all other warnings produced by the static analyzer were false positives. We simulated user interaction by repeatedly examining the alarm with highest conditional probability, and labelling it as true or false.

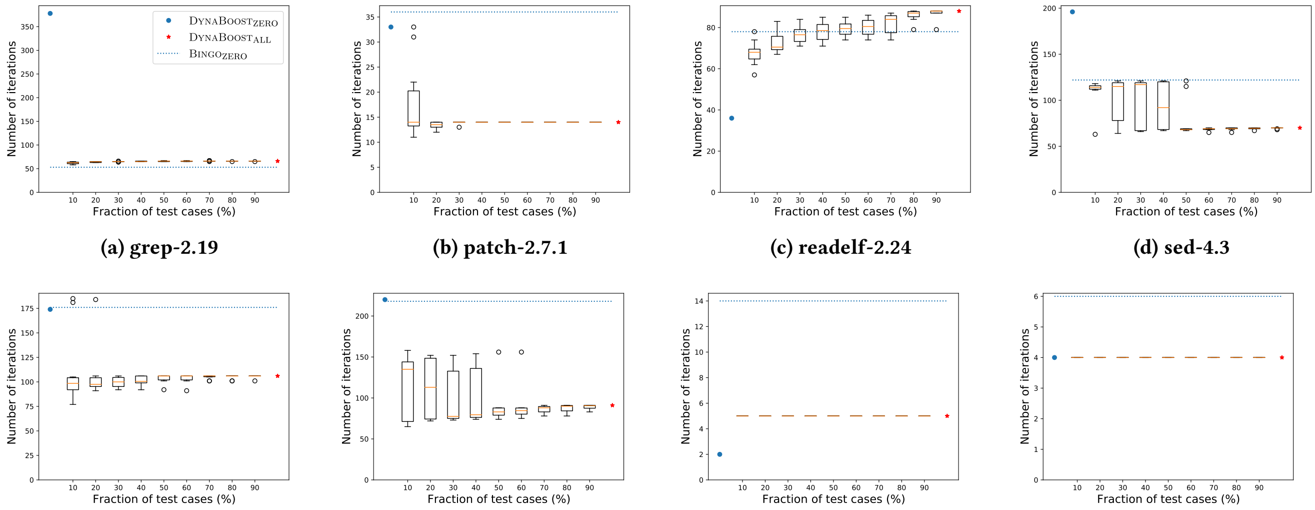
In any case, note that we provide identical labels to both DynaBoost and Bingo. Furthermore, mislabelling can affect alarms in only one direction, i.e., by mistakenly identifying real bugs as false warnings. As a result, when considering such potential mislabellings, the numbers in Table 2 provide an upper bound on the time needed to discover the first real bug using DynaBoost.

7 RELATED WORK

Dynamic analysis. A large body of research on dynamic analysis has been proposed to capture interesting properties of programs such as memory safety [19, 26, 30], datarace [7, 27], likely invariant [6]. While DynaBoost currently relies on DFSan because our underlying analyzer is based on data dependencies, we conjecture that other combinations of static and dynamic analyzers would be possible. For example, runtime information of two threads that may happen in parallel by RoadRunner [7] can be transferred to the alarm ranking system for static datarace detection [24].

Combining static and dynamic analysis. Researchers have previously investigated techniques to combine both approaches, such as by inserting dynamic checks to validate properties which are not statically provable [1, 12, 18, 29], using information collected from test executions to optimally set knobs for a subsequent analysis run [17], concolic execution to guide testing through pieces of code that are difficult to explore [25], using static analysis to minimize the amount of dynamic monitoring [3, 14], or bounded exhaustive testing [34]. Our work is different from the previous work as we combine the two approaches in a probabilistic framework for alarm ranking system.

User-guided static analysis. Previous user-guided approaches for static analysis such as alarm classification [13, 35], alarm ranking [9, 24] and alarm clustering [11] provide mechanisms to incorporate user feedback to filter out false alarms. However, human user of static analyzers, who is unaware of the details of analysis design, can provide only limited forms of feedback such as labels of alarms. DynaBoost overcomes this limitation by incorporating dynamic



(a) grep-2.19 **(b) patch-2.7.1** **(c) readelf-2.24** **(d) sed-4.3**

(e) sort-7.2 **(f) tar-1.28** **(g) latex2rtf** **(h) optipng-0.5.3**

Figure 7: Performance of DynaBoost with a limited number of test cases, sampled from the full test suite. The whisker plot indicates the distribution of observed results. The remaining plots are available in Appendix B.

analysis results from test cases, thereby boosting the performance of alarm ranking systems.

8 CONCLUSION

In this paper, we developed a probabilistic technique to leverage the results of a dynamic analysis to increase the effective accuracy of a static analyzer. By targeted instrumentation of the program, we were to able to experimentally confirm the presence of intermediate conclusions drawn by the static analyzer, and use this feedback to prioritize the generated alarms. In experiments, we demonstrated a significant reduction in the human alarm inspection burden, and improvements in other related metrics such as the quality of the initial ranking, and false generalization events. We anticipate potential applications of this research in synthesizing test cases, and in automatic fault localization.

ACKNOWLEDGMENTS

This work was partly supported by the National Science Foundation through grant CCF-2107261, and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1C1C1003876 and 2021R1A5A1021944) and Institute for Information & Communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2021-0-00758, Development of Automated Program Repair Technology by Combining Code Analysis and Mining).

REFERENCES

[1] Cyrille Artho and Armin Biere. 2005. Combined Static and Dynamic Analysis. *Electron. Notes Theor. Comput. Sci.* 131 (2005), 3–14.

[2] 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.

[3] Walter Chang, Brandon Streiff, and Calvin Lin. 2008. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security (CCS).* ACM, 39–50.

[4] Paul Eggert. 2010. *sort: fix very-unlikely buffer overrun when merging to input file.* http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=14ad7a25505ec3127cd1f07001d54d94f51f1748

[5] Dan Ellis. 2003. *Sound Examples.* https://www.ee.columbia.edu/~dpwe/sounds/

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE).* ACM, 213–224.

[7] Cormac Flanagan and Stephen Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE 2010).* ACM, 1–8.

[8] 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.

[9] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuous Program Reasoning via Differential Bayesian Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 561–575.

[10] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis.* Springer, 238–255.

[11] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Sound Non-Statistical Clustering of Static Analysis Alarms. *ACM Transactions on Programming Languages and Systems* 39, 4, Article 16 (2017), 35 pages.

[12] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. 2012. Residual investigation: predictive and precise bug detection. In *International Symposium on Software Testing and Analysis (ISSTA).* ACM, 298–308.

[13] Ravi Mangal, Xin Zhang, Aditya Nori, and Mayur Naik. 2015. A User-Guided Approach to Program Analysis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015).* ACM, 462–473.

[14] Misael Mongiovì, G. Giannone, Andrea Fornaia, Giuseppe Pappalardo, and Emiliano Tramontana. 2015. Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in java applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC).* ACM, 1573–1579.

[15] 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.

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

[17] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from tests. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL. ACM, 373–386.

[18] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, John Launchbury and John C. Mitchell (Eds.). ACM, 128–139.

[19] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM Conference on Programming Language Design and Implementation*. ACM, 89–100.

[20] 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 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM, 229–238.

[21] Hakjoo Oh, Kihong Heo, Wonchan Lee, and Kwangkeun Yi. 2012. The Sparrow Static Analyzer. https://github.com/ropas/sparrow.

[22] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

[23] Matt Powell. 2001. *The Canterbury Corpus*. https://corpus.canterbury.ac.nz/index.html

[24] 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)*. ACM, 722–735.

[25] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 263–272.

[26] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC)*. USENIX Association.

[27] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*. ACM, 62–71.

[28] 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.

[29] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 81–92.

[30] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 46–55.

[31] 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.

[32] The Clang Team. 2020. *DataFlowSanitizer*. https://clang.llvm.org/docs/DataFlowSanitizer.html

[33] Willem van Schaik. 2011. *PngSuite*. http://www.schaik.com/pngsuite/

[34] Banghu Yin, Liqian Chen, Jiangchao Liu, Ji Wang, and Patrick Cousot. 2019. Verifying Numerical Programs via Iterative Abstract Testing. In *SAS (Lecture Notes in Computer Science, Vol. 11822)*. Springer, 247–267.

[35] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective Interactive Resolution of Static Analysis Alarms. *Proceedings of the ACM on Programming Languages* OOPSLA, Article 57 (2017).