Toward More Efficient Statistical Debugging with Abstraction Refinement

ZHIQIANG ZUO, XINTAO NIU, and SIYI ZHANG, State Key Laboratory for Novel Software Technology at Nanjing University, China
LU FANG, University of California, Irvine, USA
SIAU CHENG KHOO, National University of Singapore, Singapore
SHAN LU, University of Chicago, USA
CHENGNIAN SUN, University of Waterloo, Canada
GUOQING HARRY XU, UCLA, USA

Debugging is known to be a notoriously painstaking and time-consuming task. As one major family of automated debugging, statistical debugging approaches have been well investigated over the past decade, which collect failing and passing executions and apply statistical techniques to identify discriminative elements as potential bug causes. Most of the existing approaches instrument the entire program to produce execution profiles for debugging, thus incurring hefty instrumentation and analysis cost. However, as in fact a major part of the program code is error-free, full-scale program instrumentation is wasteful and unnecessary.

This article presents a systematic abstraction refinement-based pruning technique for statistical debugging. Our technique only needs to instrument and analyze the code partially. While guided by a mathematically rigorous analysis, our technique is guaranteed to produce the same debugging results as an exhaustive analysis in deterministic settings. With the help of the effective and safe pruning, our technique greatly saves the cost of failure diagnosis without sacrificing any debugging capability.

We apply this technique to two different statistical debugging scenarios: in-house and production-run statistical debugging. The comprehensive evaluations validate that our technique can significantly improve the efficiency of statistical debugging in both scenarios, while without jeopardizing the debugging capability.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; *Dynamic analysis*; *Software maintenance tools;*

Additional Key Words and Phrases: Statistical debugging, fault localization, abstraction refinement, selective instrumentation

This article refines and extends the work published in two previous conferences, ISSTA'14 [76] and OOPSLA'16 [75].
This work was partially supported by the National Natural Science Foundation of China (Nos. 62032010 and 62102176), the Natural Science Foundation of Jiangsu Province (No. BK20191247), the US National Science Foundation under grants CNS-1613023, CNS-1703598, CNS-1763172, CNS-2006437, CNS-2007737, and CNS-2106838, and the US Office of Naval Research under grants N00014-16-1-2913 and N00014-18-1-2037.
Authors' addresses: Z. Zuo (corresponding author), X. Niu (corresponding author), and S. Zhang, State Key Laboratory for Novel Software Technology at Nanjing University, China; emails: {zqzuo, niuxintao}@nju.edu.cn, mg1933093@smail.nju.edu.cn; L. Fang, University of California, Irvine; email: lfang3@uci.edu; S. C. Khoo, National University of Singapore, Singapore; email: khoosc@nus.edu.sg; S. Lu, University of Chicago; email: shanlu@cs.uchicago.edu; C. Sun, University of Waterloo, Canada; email: cnsun@uwaterloo.ca.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2023/03-ART36 $15.00
https://doi.org/10.1145/3544790

ACM Reference format:
Zhiqiang Zuo, Xintao Niu, Siyi Zhang, Lu Fang, Siau Cheng Khoo, Shan Lu, Chengnian Sun, and Guoqing Harry Xu. 2023. Toward More Efficient Statistical Debugging with Abstraction Refinement. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 36 (March 2023), 38 pages.
https://doi.org/10.1145/3544790

1 INTRODUCTION

Bugs are prevalent in software systems before or even after deployment. As is well known, debugging is a notoriously painstaking and time-consuming task. To reduce developers' burden, researches have proposed a wide variety of automated debugging approaches. As one major family of automated debugging, statistical debugging[1] collects failing and passing executions and applies statistical techniques to identify discriminative elements as potential bug causes [1, 15, 26, 33, 38, 61]. The rationale is that program elements that are frequently executed in the failing runs but rarely executed in the passing runs are quite likely to be faulty.

1.1 Motivation

The problem with these statistical debugging approaches is that they consider *every program element* to be potentially relevant to the failure, and thus instrument and analyze the *entire program* for debugging. Such full-scale program instrumentation incurs hefty cost in terms of CPU time, memory consumption, network transfer, and disk storage, not just during instrumentation and runtime execution but also analysis thereafter. However, in fact, most of the program code works well, and *only small portions of a program are relevant to a given bug* [16]. As stated in Reference [38], the majority (often 98%–99%) of program elements (e.g., predicates) monitored are irrelevant to program failures, and thus unnecessary for instrumentation and analysis.

An open and fundamental challenge in statistical debugging is how to quickly identify a small number of failure-correlated program entities (e.g., needles) from an ocean of entities in a program of a reasonable size (e.g., a haystack), while more importantly, guaranteeing debugging quality. This motivates us to devise a selective instrumentation technique such that only the necessary program elements that are highly correlated to the failure are instrumented, ignoring irrelevant ones. As such, the user-side overhead, data transfer, storage, and analysis cost can be significantly reduced.

To address this challenge, two selective instrumentation approaches have been proposed so far. One uses the random sampling technique [37], with the hope that the predicate profiling cost can be amortized across a large number of users and runs. While random sampling reduces the overhead the user experiences in *each run* of the instrumented program, it does *not* reduce the aggregated total cost of data collection and analysis from the developer's perspective. In other words, although sampling collects less data from each run at each end-user, to achieve statistical significance, more runs/end-users need to be involved and their data need to be transferred, leading to increased latency for failure diagnosis and delayed patch design. For example, under the common 1/100 or 1/1,000 sampling rate, hundreds or thousands more failure runs need to be traced before sufficient predicates get sampled to produce statistically meaningful results [6, 29, 37]. Furthermore, a whole-program sampling infrastructure may lead to a large baseline overhead (e.g., more than 50%) that cannot be amortized through sampling [8].

[1] Some also call it statistical fault localization or spectrum-based fault localization. We treat them as the same class of approaches in this article.

The other approach uses the heuristics-guided sampling technique [7, 10, 16], with the hope that failure-correlated predicates get sampled earlier than others. Some assume that failure predictors are likely around failure points [7, 10], while others first profile predicates near the program entrance and iteratively search down the control-flow graph based on the failure-correlation metric, informally referred to as *suspiciousness* in this article, of already profiled branch predicates. Unfortunately, these heuristics-guided techniques still have limitations. Each of these heuristics naturally only works for some types of bugs and predicates and may behave worse than random sampling in other cases [7, 10]. More importantly, they can never prune out any predicates while guaranteeing the quality of debugging reports—without exhaustively profiling all predicates, it is impossible for them to know whether the best predictor has been found or not. As a result, some of these techniques simply terminate after a given number of predicates are profiled, sacrificing diagnosis ability [7]. Others require developers to *manually* and *periodically* check diagnosis results to determine whether a good enough failure predictor has been found [10], which is, obviously, a daunting task that most developers would be reluctant to do [51].

Instead of looking for a new sampling strategy, this article takes on a new quest driven by a key question: *Can we prune the predicate space with quality guarantees?* If so, then we can reduce diagnosis cost without sacrificing diagnosis ability.

1.2 Our Technique

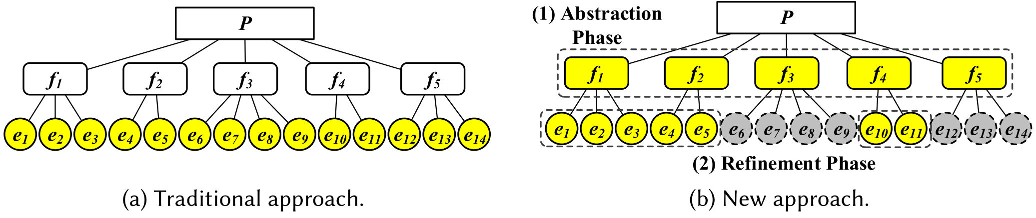
We propose a general, rigorous, and automated pruning technique for predicated statistical debugging. Our technique applies to all types of predicates. It can greatly reduce the number of predicates that need to be profiled and analyzed, while never missing top-ranked failure predictors with *mathematical guarantees*[2] under the assumption that a fixed set of test cases are used during debugging. With the help of the effective pruning, our approach greatly saves the cost of failure diagnosis without sacrificing any failure diagnosis capability.

Our technique is inspired by—and formulated as an instance of—the *abstraction refinement* framework [17]. Our key observation is simple: Predicates are *concrete* program entities constituting a huge space; profiling and analyzing them directly is doomed to result in a high cost. If we can raise the *abstraction level* by first profiling and analyzing data from coarse-grained[3] program entities (e.g., functions), then we may obtain a bird's-eye view of how each coarse-grained entity is correlated with a failure. This view may then help us decide: (1) which coarse-grained entity should be *refined*, with all the fine-grained entities (e.g., predicates) it represents profiled and analyzed; and (2) which coarse-grained entity does not need to be refined, with all the fine-grained entities it represents pruned away. Informally speaking, using inexpensive, coarse-grained suspiciousness information, we can efficiently and rigorously identify the top k fine-grained suspicious predicates—the ultimate goal of statistical debugging—by profiling and analyzing only a small portion of the large predicate space.

To carry out this insight, there are two critical questions to answer: (1) given an existing suspiciousness metric I for concrete (fine-grained) entities, how to design a suspiciousness metric C for abstract (coarse-grained) entities; and (2) how to use C to guide the fine-grained predicate profiling and analysis?

Designing Metric C. Answering these questions is not trivial. Let us first consider the first question. Our goal here is to make C as indicative of I as possible so precise refinement guidance can be obtained from applying C to coarse-grained profiles. An ideal design of C is such that the

[2] We will refer to the guarantees several times in the article and discuss the underlying assumptions in details in Section 7.1
[3] The terms "abstract" and "coarse-grained," and "concrete" and "fine-grained" are used interchangeably.



(a) Traditional approach. (b) New approach.

Fig. 1. A high-level comparison between the traditional approach and our approach.

Table 1. A Comparison between Existing Statistical Debugging Techniques and the Proposed Work

Approaches	User-side Overhead	Data Analysis Effort	IsAuto	Quality of Results
Sampling-based [37, 59]	Dep. on sampling rate	Analyze all profiles	Yes	Dep. on sampling rate
Iterative [10, 16]	Low	Analyze selected profiles	No	Dep. on heur. and manual insp.
This work	Low	Analyze selected profiles	Yes	Fully precise

function with the highest C value must contain the predicates that have the highest I values. Hence, to identify the top k predicates, one only needs to refine and analyze a small number of functions with the highest C values. However, this ideal situation is impossible to achieve, as designing such a C that works for all types of predicates is infeasible.

An alternative way to design C is to use various kinds of heuristic information so the top bug-correlated predicates (e.g., ranked by I) are *likely* to be inside the most suspicious functions (e.g., ranked by C). However, this is fundamentally no better than heuristics used by previous work.

To overcome this challenge, we take a novel perspective that allows us to explore the middle ground. Given a suspiciousness metric I for predicates, we derive C from I with the guarantee that for any function f, $C(f)$ gives the *tightest upper bound* of all $I(e)$ such that e is a predicate in f. This upper bound guarantee will be useful in guiding predicate profiling and pruning, as discussed below. The derivation is done by exploring the entire input space of I and encoding the inputs and outputs of C in a table; details of this derivation can be found shortly in Section 4.

Using C to Guide Refinement. Following our design of C, we answer the second question with a diagnosis process consisting of two major phases, with the high-level idea illustrated in Figure 1(b).

The first phase conducts simple and lightweight profiling of functions (i.e., abstract entities indicated by the yellow rectangles in Figure 1(b)), collecting a set of *abstract profiles* for functions. These profiles are then used to calculate $C(f)$ for each function f.

The second phase conducts refinement. Only a subset of functions are refined by profiling *all* predicates inside each function. For instance, the predicates corresponding to yellow circles in Figure 1(b) are profiled. Since the suspiciousness of f, $C(f)$, provides an upper bound for the I values of all predicates inside f, our diagnosis *automatically* determines a subset of functions whose $C(f)$ is no less than the top-kth I value, effectively pruning away a large number of predicates (i.e., dashed gray circles in Figure 1(b)) from profiling and analysis. The detailed discussion and formulation of the refinement process can be found in Section 3.

As a result, our approach is fully automated and fully precise: The mathematical guarantee (cf. Theorem 3.6) dictates that when our process terminates, the top bug predictors reported are the same as those that would have been reported by an exhaustive approach that instruments all predicates. In comparison, the original approach, shown in Figure 1(a), does not profile any functions. Instead, it profiles all fine-grained predicates in one phase. A quantitative comparison between our approach and several representative existing techniques can be found in Table 1.

We applied our abstraction refinement-based pruning technique to two statistical debugging scenarios: *in-house* debugging and *production-run* debugging. Briefly, in-house debugging aims to diagnose bugs appeared before software deployment. They stand for the approaches [31, 33, 61] that require developers to run the instrumented program in the lab to obtain execution profiles. While production-run debugging approaches [6, 10, 37] are performed to debug production-run failures (or field failures [22]) in deployed programs. Users run the deployed instrumented program to generate execution profiles that are sent to developers for subsequent analysis. Due to the different ways of obtaining execution profiles, their performance focus varies. For in-house debugging approaches, our goal is to reduce the instrumentation/analysis cost as well as the debugging latency to make debugging more efficient. For production-run debugging, besides the instrumentation and analysis cost spent by developers, the most crucial factor to be considered is the user-side runtime overhead. We are required to guarantee sufficiently low user-side overhead to encourage active participation from large user base. Based on the difference, we adapted our technique particularly for each scenario. For in-house debugging (Section 5), the refinement phase is simply implemented in two-pass manner where all the suspicious methods are refined within one pass, thus reducing the debugging latency. While for production-run debugging (Section 6), to ensure low enough user-side runtime overhead, an iterative procedure is adopted where the suspicious methods are refined one-by-one via multiple refinements. The detailed descriptions of each scenario will be given shortly.

1.3 Contributions

We made the following contributions:

- We propose a general and automated abstraction refinement-based pruning technique for statistical debugging and apply it to two different debugging scenarios, namely, in-house statistical debugging (Section 5) and production-run statistical debugging (Section 6).
- We design an abstract suspiciousness metric and devise the dynamic programming style algorithm to compute it. We build the rigorous mathematical relation between the suspiciousness scores of fine-grained and coarse-grained elements to ensure safe pruning.
- We conduct a rich set of experiments to validate the effectiveness of our technique. The comprehensive evaluations on both C and Java subject programs show promising results. All the code and scripts used in our evaluation can be accessed via the following link: https://github.com/k-y-zhang/statistical-debugging-with-abstraction-refinement.

1.4 Outline

The rest of this article is organized as follows: Section 2 introduces relevant background regarding predicated statistical debugging. We give the detailed description of our abstraction refinement-based pruning technique in Section 3, followed by the design and computation of abstract suspiciousness metric in Section 4. Sections 5 and 6 describe the applications of our technique to in-house debugging and production-run debugging, respectively. A series of important issues are then presented in Section 7. We discuss the related work in Section 8 and conclude in Section 9.

2 BACKGROUND

This section explains the important components of statistical debugging, including predicates, the metrics used for measuring predicates' suspiciousness, and two different debugging scenarios: in-house and production-run debugging.

2.1 Predicates

An instrumentation scheme widely used in the statistical debugging community was developed by Liblit et al. [38]. It collects the outcome of *predicates* at runtime, representing certain program state information. There are three categories of predicates considered, which have been shown to be effective for diagnosing a wide variety of software bugs:

- **Branches:** For each conditional, two predicates are tracked to indicate whether the *true* or *false* branch is taken at runtime.
- **Returns:** At each scalar-returning call site, six predicates are created to capture whether the return value r is ever $> 0, \geq 0, < 0, \leq 0, = 0$, or $\neq 0$.
- **Scalar-pairs:** Ateach assignment of a scalar value, six predicates are considered: $x \;\; <, \leq, >, \geq, =, \neq \;\; y_i$ (or c_j), where x is the assigned value, y_i and c_j represent one of the other same-typed in-scope variables and one of the constant-valued integer expressions seen in the program, respectively.

By means of software instrumentation or hardware profiling [6], a profile consisting of the runtime values of these predicates is derived for each run. Each profile contains a set of predicate counts, each recording the number of times a predicate is evaluated to true during the run. Only those predicates whose counts are not less than 1 (i.e., the predicate is evaluated to true at least once) are retained. Each profile is thus regarded as a set of items, each of which is a predicate evaluated to true at least once during execution. In addition, each profile is labeled as passing or failing, depending on whether it is collected from a passing or failing run. All the profiles constitute a labeled itemset database, whose each transaction corresponds to one profile.

2.2 Statistical Metrics

After the profiles are obtained from many runs, statistical analysis is performed to compute a *suspiciousness* value for each predicate (or a set of predicates that is termed as a bug signature [26]). The top-scored predicate (or set of predicates) is regarded as the best *predictor* of the failure.

Various metrics have been proposed and studied to measure the statistical suspiciousness [33, 41, 45]. They commonly take into consideration four parameters: namely, p, n, p_t, n_t. Given a predicate (or a set of predicates) denoted as e, $p_t(e)$, and $n_t(e)$ indicate the number of passing and failing runs in which e is observed to be true, respectively; while $p(e)$ and $n(e)$ represent the total number of passing and failing runs in which e is observed,[4] no matter true or false, respectively. All of them are natural numbers, i.e., $p \in [0, P], n \in [0, N], p_t \in [0, p], n_t \in [0, n]$, where P and N are the total number of passing and failing runs, respectively.

Here, we discuss two representative metrics, *Ochiai* [3] and *IG* **(Information Gain)** [54]. *Ochiai* origins from the study in Botany and Zoology [47], which is both theoretically and empirically validated to be a well-performed statistical debugging metric [2, 45, 53]. *IG* is adopted in bug signature identification (MPS) [61], an in-house debugging approach.

Given a predicate e, *Ochiai(e)* is formally defined as Equation (1) to measure the correlation between e and failing runs. In the application of production-run debugging discussed shortly in Section 6, *Ochiai* is used to instantiate I as the fine-grained suspiciousness metric.

$$Ochiai(e) = \frac{n_t(e)}{\sqrt{n(e) * (n_t(e) + p_t(e))}} \tag{1}$$

[4]Given a negative return value, all six returns predicates ($> 0, \geq 0, < 0, \leq 0, = 0, \neq 0$) are "observed." However, only three of them ($< 0, \leq 0, \neq 0$) are "evaluated to be true." p and n can be pre-computed easily based on p_t and n_t.

The second metric we would like to describe is IG, which can be defined as follows:

$$IG(e) = H(P, N) - \frac{p_t(e) + n_t(e)}{P + N} \times H(p_t(e), n_t(e)) - \frac{(P + N) - (p_t(e) + n_t(e))}{P + N} \times H(P - p_t(e), N - n_t(e)), \quad (2)$$

where

$$H(a, b) = -\frac{a}{a + b} \times \log_2\left(\frac{a}{a + b}\right) - \frac{b}{a + b} \times \log_2\left(\frac{b}{a + b}\right).$$

In an in-house bug signature mining approach [61], the following *discriminative significance* metric (Equation (3)) based on IG is used to measure the fine-grained suspiciousness of predicates:

$$DS(e) = \begin{cases} IG(e) & \text{if} \frac{n_t(e)}{N} > \frac{p_t(e)}{P} \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

Note that a bug signature is not restricted as a singleton predicate in Reference [61]. It is generalized to be a set of predicates. For a set of predicates $\overline{e}$, the metric works as usual as long as $p_t(\overline{e})$ and $n_t(\overline{e})$ indicate the number of passing and failing runs where all the predicates in $\overline{e}$ are observed to be true, respectively

2.3 In-house and Production-run Statistical Debugging

According to the sites where the profiles are collected, we can roughly categorize statistical debugging approaches as two different scenarios, namely, in-house debugging [15, 31, 32, 34, 61] and production-run debugging [6, 7, 10, 16, 38]. To validate the applicability of our technique, we select one representative work for each scenario—bug signature mining [61] for in-house debugging and cooperative bug isolation [38] for production-run debugging—and apply our technique to each of them. The detailed design and implementation of these applications will be given shortly in Sections 5 and 6. The following gives the description of each original approach:

In-house Bug Signature Mining [61]. To provide informative clue for identifying, understanding, and fixing bugs, more contextual information where the bug occurs is highly desired. To this end, bug signature discovery is thus proposed [15, 26, 61]. Instead of pinpointing a single suspicious element (statement or predicate), bug signature mining aims to uncover the bug context information comprising multiple elements. To enhance the predictive power of bug signatures, Sun and Khoo [61] proposed *predicated bug signature mining* (MPS for brevity), where both data predicates and control flow information are utilized.

Given the buggy program and test suites, both passing and failing profiles comprising predicates (as discussed in Section 2.1) are first collected. MPS [61] then takes the profiles as input and performs two steps, i.e., preprocessing and mining, to finally obtain predicative bug signatures. In Reference [61], the input profiles are first pre-processed to produce a dataset that is subsequently fed into the bug signature miner. To begin with, some unimportant or redundant predicates are filtered out in advance. Note that preprocessing is essential, as it constructs the database in a suitable format for the subsequent mining step; furthermore, it filters a great number of predicates to effectively reduce the scale of mining. However, it is also quite expensive especially if the profiles processed are of big size. Given a predicate itemset database constructed from profiles through preprocessing, and the number of top discriminative signatures to mine k, a discriminative itemset mining algorithm discovers the top-k discriminative bug signatures based on the *discriminative significance* measure listed as Equation (3). They provided two modes of signature mining: *inter-procedural* and *intra-procedural*. Since the *inter-procedural* signature mining where the predicates

in a signature can span across multiple functions is much more expensive than the *intra-procedural*, we focus on improving the efficiency of the *inter-procedural* mode in this work. We refer readers to Reference [61] for the detailed mining algorithm. As usual, MPS requires instrumenting the entire program to produce full-scale execution profiles for mining. Such full-scale instrumentation is unnecessary and severely undermines the efficiency of debugging.

Production-run Cooperative Bug Isolation [39]. Most software deployed around the world remains buggy in spite of extensive in-house testing. These failures that occur after deployment on user machines are called *field failures* [18]. To debug these field failures, developers are required to reproduce them in the laboratory according to the bug reports and then perform the same process as in-house debugging [30]. However, in practice, it is usually hard for developers to reproduce field failures (especially client-side failures) in the lab due to the difference in environments and configurations, as well as nondeterminism. To tackle this problem, production-run *cooperative (statistical) bug isolation* [37–39] has been proposed and received much attention. This approach applies the idea of crowd-sourcing in sampling classes of program runtime behavior from a large pool of end-users running instrumented programs for bug tracking. The gathered program traces enable developers to apply statistical techniques to pinpoint the likely causes of failures.

When each end-user runs the deployed program, a profile consisting of predicates (as discussed in Section 2.1) is recorded and sent to the developer. This happens simultaneously at a large number of end-users' sites. Once a sufficient number of profiles are collected at developer's site, a statistical metric (e.g., *Ochiai*, shown as Equation (1)) is adopted to measure the suspiciousness of each predicate. Eventually, the predicates with the top-k highest suspiciousness scores are reported as bug predictors. The success of this approach hinges on the availability of a sufficiently large user base to run the instrumented programs. To encourage a great number of users to participate, the user's overhead for running the instrumented programs should be kept sufficiently low. To this end, Liblit et al. adopted a sparse random sampling technique [37], which amortizes the cost of monitoring user-end executions to a large number of users so each user suffers a relatively low time overhead. Nevertheless, from the perspective of developers, this approach does not really reduce the monitoring cost or the total size of execution data, which consumes many resources such as network bandwidth, storage space, CPU time, due to the need for data transfer, storage, and analysis. This constrains the applicability of post-deployment debugging, especially for large applications.

3 ABSTRACTION-GUIDED STATISTICAL DEBUGGING

This section presents our abstraction-guided statistical debugging technique, first through an informal description illustrated by an example in Section 3.1 and then through an abstraction refinement-based formulation in Section 3.2.

3.1 Overview

Basic Idea. At the core of all existing approaches is predicate profiling. While the outcomes of predicates are critical for failure diagnosis, whole-program predicate tracking and handling is too expensive. As discussed in Section 1, existing overhead reduction techniques all have drawbacks. None of them can automatically reduce the overall cost of predicate tracking and handling without sacrificing diagnosis quality.

The goal of our work is to automatically prune the predicate space without sacrificing diagnosis quality, and we will achieve this through lightweight information collected for coarse-grained entities such as functions.

Two-phase Approach. Our statistical debugging has a two-phase process: an *abstract information collection* phase followed by a *refinement* phase.

In the first phase, instead of profiling predicates, our approach profiles functions, recording which function is executed at least once in each run. This profiling is lightweight, and the resulting profiles can be analyzed to obtain the *suspiciousness* score for each function based on a new metric C. We will discuss what is C and how it is designed in Section 4. Here, we just need to keep in mind that C is derived from the fine-grained predicate-level suspiciousness metric I. Given a function f, $C(f)$ is guaranteed to be the *lowest upper bound* of all such $I(e)$ that e is a predicate inside f.

The second phase conducts refinement. Given the coarse-grained suspiciousness information $C(f)$ for each function f collected in the first phase, our technique automatically determines the suspicious functions that need to be refined. Assume we are only interested in the suspicious predicates e whose $I(e) \geq \theta$ where θ indicates a constant suspiciousness threshold, as the coarse-grained suspiciousness value $C(f)$ is guaranteed to be the upper bound of all the fine-grained suspiciousness value $I(e)$ that e is a predicate inside f, only the functions f whose $C(f) \geq \theta$ needs to be refined.

Due to the distinct considerations of different debugging scenarios, the refinement phase is conducted differently. For in-house debugging, to reduce the debugging delay, a two-pass process is adopted, namely, boosting pass and pruning pass. The first boosting pass is intended to achieve a threshold θ that is then used in the subsequent refinement pass for safely pruning considerable predicates e whose $I(e) < \theta$. Specifically, the boosting pass picks the top function f that has the highest $C(f)$ value and instruments all the predicates f contains. The instrumented program is then executed using all the failing and passing test cases to acquire fine-grained profiles. By analyzing the profiles collected in boosting pass, a threshold θ corresponding to the top-kth I value of predicates is received. In the second pruning pass, predicates with I value lower than θ are pruned away, leaving behind a set of prospective predicates constituting the top-k suspicious predicates. Only these prospective predicates are instrumented in the pruning pass. Finally, analyzing the profiles collected yields the identical top-k suspicious predicates as full instrumentation. Note that, since we have obtained the fine-grained profiles corresponding to the predicates instrumented during the first pass, we do not need to instrument these predicates again in the second pass, thus further reducing execution time and storage space for profile collection.

While in production-run debugging, for the sake of sufficiently low user-side runtime overhead, an iterative refinement process is applied so only one function is refined at each iteration. The threshold θ is also updated iteratively. Each iteration retrieves the top function f from the list $\mathcal{L}$ of all not-yet-refined functions ranked by their suspiciousness values in C and *refines* f by instrumenting *all* predicates it contains. Note that our instrumentation is *not* accumulative—the instrumentation code added in previous iterations is all removed. The re-instrumented program is executed and, similar to the original approach, I is employed to produce a suspiciousness value for each instrumented predicate. At this point, suspiciousness values have been obtained for predicates instrumented in both the current iteration and all previous iterations. These predicates are then sorted and the top k predicates are identified. At the end of each iteration, we need to decide whether the existing top k predicates are already the top k predicates the original (exhaustive) approach would have produced globally. This decision is made by comparing the lowest suspiciousness value $I(e)$ among the existing top k predicates with the highest abstract suspiciousness value $C(f')$ among the remaining unrefined functions in $\mathcal{L}$:

- If $I(e) > C(f')$, then the debugging process can be safely terminated with the guarantee that the globally top k predicates have been found. The reason is that $C(f')$, which is smaller than $I(e)$, is an upper bound for the suspiciousness value of all the remaining unrefined functions

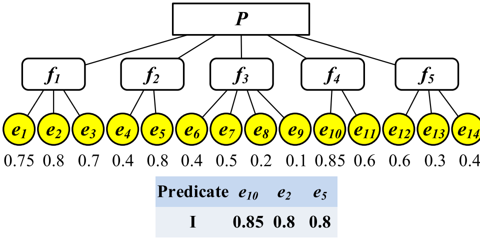


Fig. 2. The original approach.

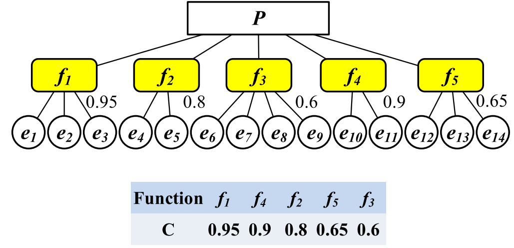
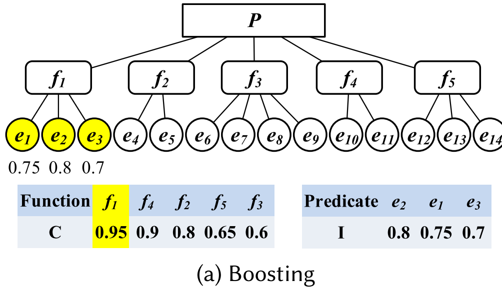
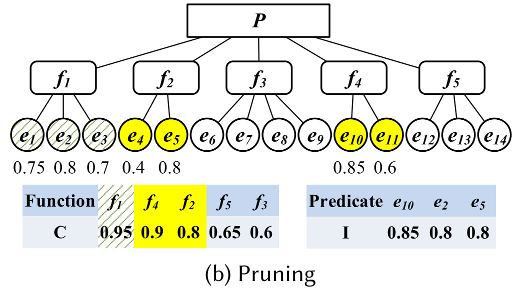


Fig. 3. The abstraction phase.



(a) Boosting



(b) Pruning

Fig. 4. The two-pass refinement phase for in-house debugging: (a) boosting and (b) pruning.

and the non-profiled predicates within them. Consequently, all the non-profiled predicates can be pruned at this point without affecting diagnosis quality.

- If $C(f') \geq I(e)$, then we will enter the next refinement iteration, as there might be a predicate in method f' with a suspiciousness value equal or higher than $I(e)$.

Example. We use an example shown in Figures 2–5 to illustrate the difference between the traditional approach and our approach. Here, a program P is represented as a tree structure where leaf nodes e represent predicates and non-leaf nodes f represent functions.

The original statistical debugging technique (shown as Figure 2) profiles all predicates across the whole program through many runs and computes a suspiciousness value I for each predicate. Predicates are ranked based on their suspiciousness, and the top k predicates are reported. In the example shown in Figure 2, each predicate has a hypothetical suspiciousness measurement and the top 3 ($k = 3$) suspicious predicates are shown in the bottom table.

Our debugging technique is different. The abstraction phase is illustrated in Figure 3. This phase profiles every function. The values of the C metric obtained after this phase are shown next to the function nodes. The refinement phase is performed differently for different scenarios.

For *in-house statistical debugging* shown as Figure 4, the refinement phase starts with a boosting pass retrieving the top suspicious function f_1 and instrumenting e_1, e_2, and e_3 in f_1, as shown in Figure 4(a). The profiles collected by running the re-instrumented program give rise to the following I values: 0.8 (e_2), 0.75 (e_1), and 0.7 (e_3). We take $I(e_3)$ (i.e., 0.7) as the threshold and get a set of candidate functions (i.e., f_4 and f_2) whose C is higher than the threshold (i.e., 0.7). Subsequently, a pruning pass is performed and all predicates in these selected functions are instrumented. The refinement pass shown in Figure 4(b) profiles all the predicates in f_4 and f_2. Together with suspiciousness values already obtained from the boosting pass, a re-ranking identifies the new top three predicates: e_{10}, e_2, and e_5. Clearly, this report is the same as what the exhaustive approach would

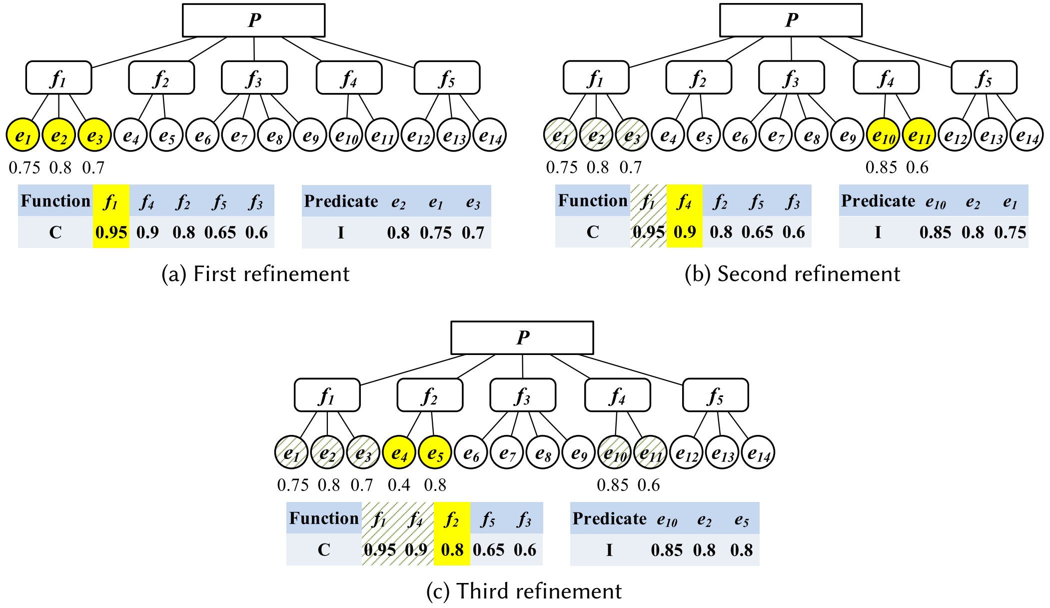


Fig. 5. The iterative refinement phase for production-run debugging: (a) first refinement, (b) second refinement, and (c) third refinement.

have produced, although only predicates in three of the five functions are instrumented, profiled, and analyzed.

In *production-run statistical debugging*, for the sake of low user-side overhead, an iterative refinement phase demonstrated as Figure 5 is employed. The refinement phase starts with its first refinement iteration that picks the top suspicious function f_1 and instrumenting e_1, e_2, and e_3 in f_1, as shown in Figure 5(a). Collecting and analyzing the profiles in the first refinement give rise to the following I values: 0.8 (e_2), 0.75 (e_1), and 0.7 (e_3). To know whether e_2, e_1, and e_3 are the top three predicates, we compare $I(e_3)$ (i.e., 0.7) with $C(f_4)$, because f_4 has the next highest suspiciousness value. Since $0.9 > 0.7$, f_4 may contain predicates whose I is higher than that of e_3 (i.e., the third predicate identified) and thus another iteration of refinement is needed. Hence, all predicates in f_4 are instrumented. The second iteration shown in Figure 5(b) profiles the two predicates in f_4. Together with suspiciousness values already obtained from the previous iteration, a re-ranking identifies the new top three predicates: e_{10}, e_2, and e_1. Similarly, we compare the $I(e_1)$ with $C(f_2)$. Since $I(e_1) = 0.75 < C(f_2) = 0.8$, the third refinement (Figure 5(c)) is required. At the end of the third refinement, the top three predicates are updated to be e_{10}, e_2, and e_5. Because the next function on the remaining list is f_5 and $I(e_5)$ (i.e., 0.8) is greater than $C(f_5)$ (i.e., 0.65), we are guaranteed that e_{10}, e_2, and e_5 must be the globally top three predicates and the debugging process can be safely terminated. Clearly, the same results are reported as what the exhaustive approach would have produced.

3.2 Problem Formulation

After the above informal description, we now present an abstraction-refinement based formulation of our technique.

Definition 3.1 (Abstract and Concrete Entities). A program P consists of a set of functions F, each of which is an *abstract entity* for instrumentation. A function $f \in F$ contains a set of predicates

E, each of which is a *concrete entity* for instrumentation. An abstraction relation $\alpha : E \rightarrow F$ maps each predicate $e \in E$ to a function $f \in F$ containing e. A concretization relation $\gamma : F \rightarrow 2^E$ maps each f to a set of predicates inside f.

In our definition, both the abstraction and concretization functions are straightforward—they are determined by the "containing" relation between functions and predicates. Although this article focuses on a two-level abstraction hierarchy, one can easily extend our technique to multiple levels, for example, by incorporating other types of entities such as loops and basic blocks. Based on the definitions of abstract and concrete entities, we define dynamic profiles.

Definition 3.2 (Abstract and Concrete Profiles and Measures). A concrete entity profile $\overline{e} \in \overline{E}$ is a five-tuple $\langle e, p(e), n(e), p_t(e), n_t(e)\rangle$, where $p(e), n(e), p_t(e), n_t(e)$ have the same meanings as introduced in Section 2.2. An abstract entity profile $\overline{f} \in \overline{F}$ is a triple $\langle f, p(f), n(f)\rangle$. Given a concrete metric I (e.g., *Ochiai* or *Information Gain*), a **concrete suspiciousness measure (CSM)** Ψ of a program is a pair (S_e, I) defined over a set of concrete entity profiles $S_e \in 2^{\overline{E}}$ reversely ordered by $I(p(e), n(e), p_t(e), n_t(e))$[5] for each $\overline{e} \in S_e$. An **abstract suspiciousness measure (ASM)** Φ of a program is a pair (S_f, C) defined over a set of abstract entity profiles $S_f \in 2^{\overline{F}}$ reversely ordered by $C(p(f), n(f))$ for each $\overline{f} \in S_f$ where C is the abstract metric.

A profile is a statistical record of an entity collected from multiple runs of the program. A concrete profile of a tracked predicate e contains four values $p(e), n(e), p_t(e)$, and $n_t(e)$ that are fed to metric I to obtain the suspiciousness value for e. An abstract profile of a tracked function f contains two values $p(f)$ and $n(f)$, representing the occurrences of f in passing and failing runs, respectively. We do not track functions' return values. Consequently, $p_t(f)$ and $n_t(f)$ are not defined.

As shown in the motivating example, at the heart of our refinement-based technique is the metric C. It takes as input two parameters $p(f)$ and $n(f)$ and returns the suspiciousness value of function f. A CSM is essentially a list of predicates ranked by their I values, while an ASM is a list of functions ranked by their C values. Before describing how C is obtained in Section 4, we first discuss here some important mathematical properties we want C to have.

Definition 3.3 (Abstraction Soundness). An ASM $\Phi = (S_f, C)$ is a sound abstraction of a CSM $\Psi = (S_e, I)$ iff $\forall \overline{e} \in S_e : \exists \overline{f} \in S_f : (1)\ (e, f) \in \alpha \wedge (2)\ C(p(f), n(f)) \geq I(p(e), n(e), p_t(e), n_t(e))$.

The second property indicates that $C(p(f), n(f))$ must be an upper bound of the suspiciousness values for all predicates e inside f. Clearly, the definition of C plays a central role in the development of a sound abstraction. Given an appropriately defined C, we can easily obtain a sound ASM Φ by instrumenting all function entries and collecting function profiles (in the first phase). The subsequent refinement phase is essentially a process of incrementally building the CSM Ψ. Our hope is that with the guidance of Φ, we can find the top k predicates without constructing the complete Ψ.

LEMMA 3.4 (UPPER BOUND GUARANTEE). *Let ASM $\Phi = (S_f, C)$ be a sound abstraction of CSM $\Psi = (S_e, I)$. Let $\overline{e_i} = \langle e, \ldots\rangle$ and $\overline{f_i} = \langle f, \ldots\rangle$ be the ith concrete and abstract entity profile in the (reversely ordered) set Ψ and Φ, respectively. We have the following guarantee:* $\forall$ *index* $i \in [0, |\Psi|), j \in [0, |\Phi|) : I(\overline{e_i}) \geq C(\overline{f_j}) \implies \forall$ *index* $k \geq j : \forall$ *index* $t \in [0, |\Psi|) : (\overline{e_t}.e, \overline{f_k}.f) \in \alpha : t \geq i$.

[5] For certain metrics such as IG, only $p_t(e)$ and $n_t(e)$ are used.

Proof. Step (1): $I(\overline{e_i}) \geq C(\overline{f_j})$ implies that, for any index $k \geq j$, $I(\overline{e_i}) \geq C(\overline{f_k})$ due to $C(\overline{f_j}) \geq C(\overline{f_k})$.

Step (2): Since ASM $\Phi = (S_f, C)$ is a sound abstraction of CSM $\Psi = (S_e, I)$ and $(\overline{e_t}.e, \overline{f_k}.f) \in \alpha$, based on Definition 3.3, we have $C(\overline{f_k}) \geq I(\overline{e_t})$.

From Steps (1) and (2), we have $I(\overline{e_i}) \geq I(\overline{e_t})$, which indicates that e_t must have a larger index (i.e., t) than e_i (i.e., i) in the CSM. □

Informally, if the suspiciousness value of a predicate e indexed i in the CSM (i.e., $I(\overline{e_i})$[6]) is ≥ the suspiciousness value of a function f indexed j in the ASM (i.e., $C(\overline{f_j})$), then as long as Φ is a sound abstraction of Ψ, the suspiciousness value of any predicate e' in f or any function lower than f in the ASM must be ≤ that of e. Therefore, e' must have a larger index (i.e., t) than e (i.e., i) in the CSM.

Definition 3.5 (Abstraction Refinement). Given a partially constructed CSM $\Psi = (S_e, I)$ and an abstraction $\Phi = (S_f, C)$, an abstraction refinement $\sqsubseteq$ produces another pair of CSM Ψ' and ASM Φ' by (1) removing the top function f from Φ, (2) instrumenting all predicates e in f, (3) executing the newly instrumented program, and (4) collecting a set of concrete entity profiles $\overline{e}$ and adding them into Ψ.

A refinement shrinks an existing ASM by removing its top entry while growing an existing CSM by adding new predicate profiles and re-sorting all contained profiles based on their I values. It is important to note that, at any step during the refinement, the entities in Φ and Ψ are always *disjoint*: For each function profile in Φ, its corresponding predicate profiles must *not* be in Ψ. In other words, Φ *soundly* abstracts the *complement* of Ψ. The refinement process starts with a full Φ and an empty Ψ and gradually removes abstract entries from Φ and adds concrete entries into Ψ.

Theorem 3.6 (Termination Quality Guarantee). *Suppose a chain of i refinement steps results in a partial CSM $\Psi = (S_e, I)$ and a partial ASM $\Phi = (S_f, C)$. For a given integer k, if $I(\overline{e_k}) > C(\overline{f_0})$, then $I(\overline{e_k})$ is guaranteed to be greater than the I values of all predicates in the functions in Φ.*

Proof. Based on Lemma 3.4, if $I(\overline{e_k}) > C(\overline{f_0})$, then for all such e_t that $(\overline{e_t}.e, \overline{f_q}.f) \in \alpha$ and $q > 0$, we have $t > k$. In other words, the index of any predicate e_t in a function in Φ must be greater than the index of e_k (i.e., k) in Ψ. Therefore, $I(\overline{e_k})$ is guaranteed to be greater than the I values of all predicates in the functions in Φ. □

Suppose our goal is to find the top k predicates that are most indicative of a bug. This theorem provides a guarantee that if the I value v of the kth predicate in the CSM is > the C value of the first function profile in the ASM Φ, then v must be > the C value of any function profile in Φ. Since the C value of a function is an upper bound of the I values of all predicates in the function, the theorem further guarantees that none of the functions yet to be refined may contain a predicate that might make the top-k list in the CSM.

4 ABSTRACT SUSPICIOUSNESS METRIC DESIGN

As we have seen, the abstract suspiciousness metric C plays a critical role in our debugging approach. This section discusses the design and computation of this metric.

[6]For simplicity of presentation, we use $I(\overline{e_i})$ and $C(\overline{f_i})$ instead of their full notations (with four and two parameters, respectively).

4.1 Properties and Design Goals

When processing profiles from a set of runs, C takes as input two parameters $p(f)$ and $n(f)$, the number of passing runs and failing runs that have executed function f. It returns the suspiciousness value of f and has to satisfy several key properties:

- Upper bound. As discussed in Section 3, requiring $C_f \geq I_e$ for all e inside f is a necessary condition to guarantee the correctness of our debugging process.
- Tightness. C has to be a tight upper bound to effectively prune the predicate space. For example, making C always return ∞ would turn our technique into an exhaustive approach. The tighter bound C gives, the more efficient our debugging process is.
- Generality. C is expected to work for all types of predicates, including branch predicates, return predicates, and scalar-pair predicates discussed in Section 2.1.

Achieving these goals is challenging, as C needs to be computed without profiling any predicates. In the following, we first present our insights around two key design questions before we present the algorithm of computing C.

What Is the Input Domain of I_e? We first discuss how to infer the input domains of I_e based on function profiles. As discussed in Section 2, I_e takes in four input parameters: $p(e)$ and $n(e)$, which denote the numbers of passing and failing runs that have observed predicate e, as well as $p_t(e)$ and $n_t(e)$, which denote the numbers of passing and failing runs that have observed e to be true.

Answers to this question hinge upon the relationship between the number of runs x that have observed a function f and the number of runs y that have observed a predicate e, among a given set of runs. We only care about whether or not f or e has been observed, not the exact number of times it has been observed in a run as discussed in Section 2. Consequently, y has to be less than or equal to x when e is inside f—if a predicate is observed, its enclosing function must have been observed. This holds for all types of predicates.

This simple observation allows us to infer the input domain of I_e based on the profile of f, when e is in f. That is, when function f is observed in $p(f)$ passing runs and $n(f)$ failing runs, the numbers of passing and failing runs that observe e must be $p(e) \in [0, p(f)]$ and $n(e) \in [0, n(f)]$, respectively. The numbers of passing and failing runs in which e is observed to be true, respectively, must be $p_t(e) \in [0, p(e)]$ and $n_t(e) \in [0, n(e)]$.

How to Compute the Upper Bound of I_e? Knowing the above relationship, ideally, we could directly derive the mathematical formula of C from the mathematical formula of I, e.g., by computing an equation modeling the area enclosed by the tangent lines of the I function. However, since I is a complex function, this derivation requires significant mathematical development and may yield large over-approximations at various points.

One might wonder if the upper bound can be computed by setting the parameters to their bound values. The answer is no, because I is *not* a monotone function. Hence, using bound values as its parameters does *not* guarantee to generate the maximum I value.

Here, we take a different and more practical approach. Given a function profile $\langle f, p(f), n(f) \rangle$, we can enumerate all valid input combinations for I_e, where e is in f, using the input-domain relationship discussed above. We can then take the maximum of these I values as C_f. This high-level idea allows us to compute C_f based on the function profiles collected in our first debugging phase *before* any predicate is instrumented or monitored.

To illustrate, consider a simple example where a function f is executed in one passing run and two failing runs (i.e., $p(f) = 1, n(f) = 2$). e is a predicate inside the function f. Without profiling e, we do not know e is observed in how many passing runs (i.e., $p(e)$) and how many failing runs (i.e., $n(e)$). Fortunately, we know that e cannot be observed in a run unless f is executed in that

run. Consequently, we can infer that e is observed in at most one passing run and two failing runs (i.e., $0 \leq p(e) \leq p(f) = 1, 0 \leq n(e) \leq n(f) = 2$). Hence, we enumerate all possible combinations of $p(e)$ and $n(e)$, namely, $(0,0)$, $(0,1)$, $(0,2)$, $(1,0)$, $(1,1)$, and $(1,2)$, and compute an I value for each of these pairs. Finally, the maximum of all these I values is used as the C value for function f. Obviously, this value is an upper bound of the I values of all predicates e inside f. Moreover, this upper bound is the lowest upper bound that can be obtained.

4.2 Our Algorithm

This subsection describes our algorithm to implement this idea. A naive algorithm would separately compute C_f for every function, which is extremely time-consuming for large software. Our design is much more efficient.

Our algorithm aims to produce a table that maps (i, j) to $C(i, j)$, where $0 \leq i \leq P$ and $0 \leq j \leq N$ with P and N being the total number of passing runs and failing runs. With this table, we can get the C value for every function f through a simple table lookup using index $(p(f), n(f))$.

To efficiently produce this table, we leverage dynamic programming[7] and an insight that the computation of $C(i, j)$ can be largely simplified by leveraging that of $C(i-1, j)$ and $C(i, j-1)$. Algorithm 1 shows the details. Following the high-level idea discussed above, Algorithm 1 computes $C(i, j)$ by enumerating and getting the maximum of all such $I(a, b, l, k)$ that $0 \leq a \leq i, 0 \leq b \leq j, 0 \leq l \leq a$, and $0 \leq k \leq b$. Line 10 shows the key optimization in our algorithm: To compute $C(i, j)$, we only need to take the maximum of already-computed $C(i-1, j)$, $C(i, j-1)$ and the to-be-computed *maxInLastTwoD*(i, j). The auxiliary function *maxInLastTwoD* takes two parameters i and j and computes the maximum value of all $I(i, j, l, k)$ where $0 \leq l \leq i$ and $0 \leq k \leq j$. This whole algorithm has an $O(P^2N^2)$ time complexity.

The amount of time and space used by Algorithm 1 is reasonable even for large programs, thanks to the benefits from dynamic programming. Moreover, in practice the number of test cases (i.e., P and N) are fixed and often limited to a maximum of hundreds or a few thousands. Furthermore, this algorithm only incurs a one-time cost: Invoking it once would generate the whole metric-matrix $C(0..P, 0..N)$. After that, the C metric for every function can be obtained by a constant-time matrix look-up.

When to Run This Algorithm? Algorithm 1 can be performed without any knowledge of the targeting software or function; the resulting C-table is valid for all types of software and functions. Consequently, we can re-use the C-table from one debugging process to another and keep expanding the table when facing larger numbers of passing (P) or failing runs (N).

In our debugging framework, after the abstract information collection phase, we will use the profile of each function f to look up the C-table and obtain the suspiciousness value of f, and then use that to conduct further refinement.

THEOREM 4.1 (LOWEST UPPER BOUND). *For each abstract entity profile* $\overline{f} = \langle f, a, b \rangle$, *a lookup* $C(a, b)$ *in the table computed by Algorithm 1 returns the* lowest upper bound *of* $I(i, j, l, k)$ *for all such concrete entity profile* $\overline{e} = \langle e, i, j, l, k \rangle$ *that* $(e, f) \in \alpha$.

PROOF. This theorem can be proved by contradiction. Based on the way C is computed in Algorithm 1, $C(a, b)$ must be equal to a particular $I(i, j, l, k)$ where $0 \leq i \leq a, 0 \leq j \leq b, 0 \leq l \leq i$, and $0 \leq k \leq j$. If $C(a, b)$ is not the lowest upper bound, then there must exist another upper bound less than $C(a, b)$ and the $I(i, j, l, k)$ from which $C(a, b)$ is obtained, contradicting its upper bound property. □

[7] A natural and efficient way to compute a mathematical property over a large (e.g., multi-dimensional) domain.

ALGORITHM 1: Computation of the abstract suspiciousness metric C.

```
Input: Metric I, the total numbers of passing and failing runs P and N obtained from the first phase
Output: C encoded as a table
   // Base case
1  C(0, 0) ← I(0, 0, 0, 0)
2  for i ← 1 to P do
3  |  C(i, 0) ← max{maxInLastTwoD(i, 0), C(i − 1, 0)}
4  end
5  for i ← 1 to N do
6  |  C(0, i) ← max{maxInLastTwoD(0, i), C(0, i − 1)}
7  end
   // Iterative computation
8  for i ← 1 to P do
9  |  for j ← 1 to N do
10 |  |  C(i, j) ← max{maxInLastTwoD(i, j), C(i − 1, j), C(i, j − 1)}
11 |  end
12 end

13 Function maxInLastTwoD(i, j)
14 largest ← 0
15 for l ← 0 to i do
16 |  for k ← 0 to j do
17 |  |  largest ← max{largest, I(i, j, l, k)}
18 |  end
19 end
20 return largest
```

The suspiciousness of a function being the lowest upper bound of the suspiciousness of its contained predicates provides a basis for the early termination of the refinement process, yielding debugging algorithms that are both precise and efficient.

It is important to note C is computed when the *entire input space* of I is considered. In practice, many integer inputs we consider when computing C may not actually appear in a particular set of predicate profiles. For these profiles, there may be a gap between the suspiciousness of a function given by C and the maximum suspiciousness of predicates in the function computed from I.

5 APPLICATION 1: IN-HOUSE STATISTICAL DEBUGGING

We demonstrate the effectiveness of our abstraction-guided pruning technique in this section by applying it to an in-house debugging scenario, particularly predicated bug signature mining [61], which is discussed in Section 2.3. As revealed by Reference [36], the efficiency of fault localization indeed affects the adoption by the practitioners. The original approach requires instrumenting the entire program to produce execution profiles for mining, severely undermining its efficiency and thus affecting its adoption in practice. Based on our pruning technique, we design and implement an efficient predicated bug signature mining approach that results in significant savings in instrumentation effort and substantial speedup in the subsequent analysis process. We abbreviate the original predicated bug signature mining [61] as **MPS (Mining Predicated Bug Signatures)**, whereas our approach as **ARMPS (MPS with Abstraction Refinement)**.

5.1 Design and Implementation

Given a buggy program and two groups (failing and passing) of test cases, the objective is to efficiently mine the top-k bug signatures that are highly correlated to the bug as measured by the *DS* values (shown as Equation (3) in Section 2.2). We here design and implement an efficient predicated bug signature mining approach based on our **abstraction refinement-based pruning technique (ARMPS)**. As discussed in Section 3.1, ARMPS comprises two phases: the abstraction phase followed by a two-pass refinement phase. Briefly, the workflow of ARMPS is as follows:

(1) Instrument function entries, run the instrumented program, and collect function profiles;
(2) Compute C with the *DS* metric together with the numbers of passing and failing runs P and N using Algorithm 1;
(3) Compute the abstract suspiciousness measure Φ (i.e., a list of functions reversely ordered by their respective C values) using C on the collected function profiles;
(4) First, refine a few functions denoted by γ (e.g., 5%) from Φ to build the initial concrete suspiciousness measure Ψ_{init} (i.e., a list of signatures reversely ordered by their respective *DS* values) to obtain the top-kth *DS* value θ;
(5) Second, refine all other functions from Φ whose $C \geq \theta$ to build the final concrete suspiciousness measure Ψ_{final}.

Algorithm 2 gives the pseudo code of ARMPS. At the abstraction phase, only *function entries* of the subject program are instrumented (Line 1). This sparsely instrumented program is then run against all the failing and passing test cases to collect the abstract profiles (Line 2). Each abstract profile consists of a set of functions that are executed at least once during execution. Given the suspiciousness metric *DS* and total numbers of failing and passing runs P and N, we compute the abstract metric C according to Algorithm 1 (Line 3). Based on C, an abstract analysis is performed on the abstract profiles to build the abstract suspiciousness measure Φ, i.e., a list of functions reversely ordered by their respective C values (Line 4). In the following, this abstract measure Φ will be utilized to guide the refinement.

The refinement phase has two passes, each of which performs predicate-level instrumentation followed by bug signature mining. The first pass is called "boosting." It aims to identify a threshold that will be used as a lower bound of the ultimate top-kth *DS* value of bug signatures. Specifically, we first refine a few functions—we select 5% of functions (i.e., $\gamma = 5\%$) in our experiments—with the highest C values in Φ (Line 5) by instrumenting all the predicates inside (Line 6). The instrumented program is executed over the test suite to acquire concrete profiles (Line 7). The profiles are preprocessed to create the mining dataset (Line 8), which is fed into the bug signature miner to build the initial concrete suspiciousness measure Ψ_{init} (i.e., an initial list of top-k bug signatures reversely ordered by their respective *DS* values) (Line 9). Note that Lines 6–9 indicate the traditional procedure of predicated bug signature mining stated by Reference [61], making it possible for modular plug-in of new debugging algorithm. As θ indicates the top-kth *DS* value in Ψ_{init}, the top-kth *DS* value in the final Ψ_{final} must be equal to or greater than θ. As such, the predicates within functions whose C value falls below θ can be safely exempted from refinement, since they cannot contribute a bug signature whose *DS* value is equal to or greater than θ. The pruning pass only refines the functions whose C values are no less than θ (Line 11). Lines 12–15 perform the bug signature mining as usual. Finally, the identical top-k bug signatures as mined by Reference [61] are returned (Line 16). Note that, since we have obtained the concrete profiles corresponding to the predicates in *boost* during the first pass, we only need to instrument other predicates in *prospect−boost* in the second pass to further reduce execution time and storage space for profile collection (Line 12). However, we have to preprocess all the profiles (i.e., *PP*+*BP*) to perform *inter-procedural* signature mining (Line 14).

ALGORITHM 2: Predicated Bug Signature Mining with Abstraction Refinement (ARMPS)

Input: buggy program G, test suite T, number of signatures mined k, percentage of functions refined for boosting γ
Output: top k suspicious signatures Ψ_{final}

```
/* abstraction phase: coarse-grained instrumentation and analysis */
```
1 Instrument all function entries in the entire program G;
2 Run all the failing and passing tests in T to collect abstract profiles;
3 $C \leftarrow$ compute the abstract suspiciousness metric using the DS metric;
4 $\Phi \leftarrow$ build the abstract suspiciousness measure using C on the collected abstract profiles;

```
/* refinement phase: fine-grained instrumentation and analysis */
// first pass: boosting
```
5 $boost \leftarrow$ *RefineForBoosting*(Φ, γ);
6 Instrument all predicates in *boost*;
7 Run all the failing and passing tests in T to collect concrete profiles BP;
8 $BD \leftarrow$ *Preprocess*(BP);
9 $\Psi_{init} \leftarrow$ *MineBugSignatures*(BD, k);
10 $\theta \leftarrow$ the kth top DS value of signatures in Ψ_{init};

```
// second pass: pruning
```
11 $prospect \leftarrow$ *RefineForPruning*(Φ, θ);
12 Instrument all predicates in $prospect - boost$;
13 Run all the failing and passing tests in T to collect concrete profiles PP;
14 $PD \leftarrow$ *Preprocess*($PP + BP$);
15 $\Psi_{final} \leftarrow$ *MineBugSignatures*(PD, k);
16 **return** Ψ_{final};

5.2 Experimental Evaluation

5.2.1 Benchmarks. We applied our technique to both C and Java programs. We adopted two repositories, i.e., SIR[8] [21] and Defects4J[9] [35], as our benchmark candidates, which are the well-known and widely used benchmarks containing both buggy programs and test suites generated both manually and automatically. To select benchmarks from the SIR repository, we first found all programs that have at least 5,000 lines of code and then removed those whose test suites contain less than 100 test cases. This left us 12 programs, among which, one could not compile and two did not have failing runs. Removing those three resulted in the selection of 9 programs. For Defects4J, we included all the subject programs with all their versions in it. Table 2 lists the subject programs, the corresponding repositories, the number of faulty versions, lines of code (i.e., LoC), the number of functions, the number of instrumentation predicates, the size of test suite used, and the language. Each program has multiple *faulty versions*. Each faulty version contains a distinct bug, which is either a real bug or manually injected.

The three types of predicates introduced in Section 2.1 (i.e., branches, returns, and scalar pairs) are considered in our evaluation. To instrument C programs, we employed the *sampler-cc* tool developed by Liblit et al. [37]. For Java, we implemented our own instrumenter named JSampler[10]

[8] Available at: https://sir.csc.ncsu.edu/.
[9] The specific version of Defects4J we chose is 1.5, available at: https://github.com/rjust/defects4j/releases/tag/v1.5.0.
[10] https://github.com/z-zhiqiang/JSampler.

Table 2. Characteristics of the Chosen Subject Programs

Subject	Repository	Faulty versions	LoC	Functions	Predicates	Tests	Language
space	SIR	34	6,199	136	25,449	1,248	C
sed	SIR	4	14,427	112	101,233	363	C
gzip	SIR	13	5,680	91	179,962	213	C
grep	SIR	5	10,068	129	228,498	809	C
bash	SIR	6	59,846	1,458	928,566	300	C
nanoxml	SIR	22	7,646	552	5,128	236	Java
siena	SIR	3	6,035	249	19,130	567	Java
ant	SIR	4	80,500	8,863	203,576	149	Java
derby	SIR	3	503,833	28,887	1,389,976	258	Java
lang	Defects4J	65	18,487	2,220	84,378	1,823	Java
chart	Defects4J	26	85,074	7,489	558,336	1,815	Java
math	Defects4J	106	50,651	4,819	845,426	216	Java
time	Defects4J	27	27,259	3,881	93,138	3,917	Java
mockito	Defects4J	38	8,901	1,317	9,218	1,141	Java
closure	Defects4J	176	83,721	9,467	199,186	3,460	Java

based on the Soot framework.[11] The machine on which the experiments were run has an Intel Xeon 3.60 GHz CPU and 32 GB main memory, running 64-bit Fedora 19.

Since the goal of this work is *not* to increase the precision of any debugging method, but rather to improve performance with quality guarantees, our evaluation focuses on understanding various performance factors in different steps of MPS [61], including instrumentation, profile collection, preprocessing, and mining. Since each instrumentation is only performed once and then the instrumented program can be run forever, its cost is not significant compared with the other steps (i.e., profile collection, preprocessing, and mining). In the following, we mainly discuss the performance improvement during the other three steps. Specifically, we measure the improvement of our approach (ARMPS) in reducing execution time and storage space during profile collection in Section 5.2.2. In Section 5.2.3, we compare with MPS [61] in terms of time and memory consumption for preprocessing and signature mining.

5.2.2 *Profile Collection.*

In this step, we run the failing and passing test cases to collect profiles. This corresponds to Lines 2, 7, and 13 in Algorithm 2. Here, we consider two aspects of performance cost, namely, the execution time for running the instrumented programs and the disk storage space used for profiles.

As both performance costs are to some extent dependent on the number of predicates instrumented, we first present the percentage of predicates instrumented by our approach. Figure 6 depicts the percentage of predicates instrumented totally during both boosting and pruning to obtain the top k bug signatures for each subject program. Note that as reported by an empirical study [36], over 70% of developers only check top-5 ranked elements; nearly 98% of respondents agreed that inspecting more than 10 program elements is beyond their acceptability level. Based on this, we set k as 1, 5, and 10 to assess how well our technique performs. As can be seen, compared with MPS where all the predicates (100%) in the whole program are instrumented, ARMPS manages to prune away considerable predicates. Overall, more than half of the predicates are exempted from instrumentation compared with MPS. Note that ARMPS performs quite well especially on large programs. For subjects *ant*, *derby*, *lang*, *chart*, *math*, *time*, and *closure*, even

[11] http://sable.github.io/soot/.

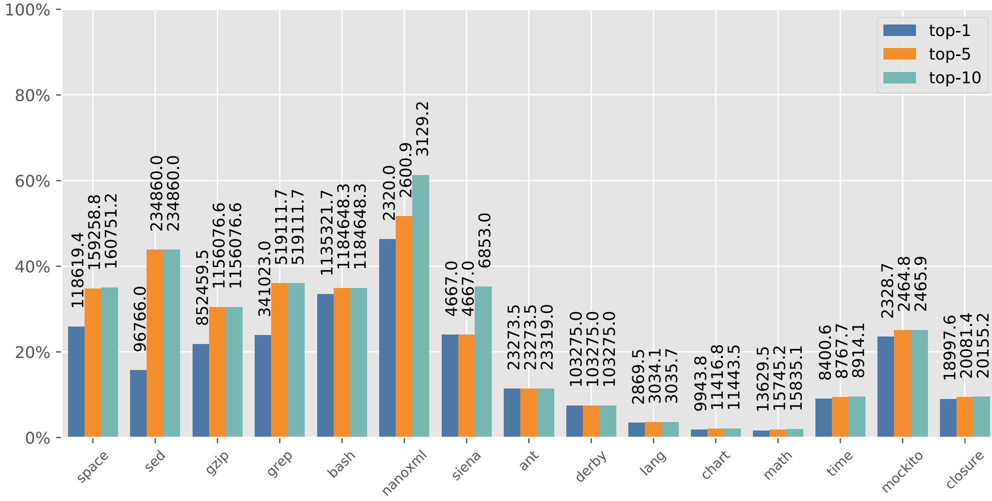


Fig. 6. Percentage and detailed number of predicates instrumented compared with 100% full instrumentation in MPS [61]: The y-axis indicates the percentage, while the number of predicates with one decimal are shown on top of each bar.

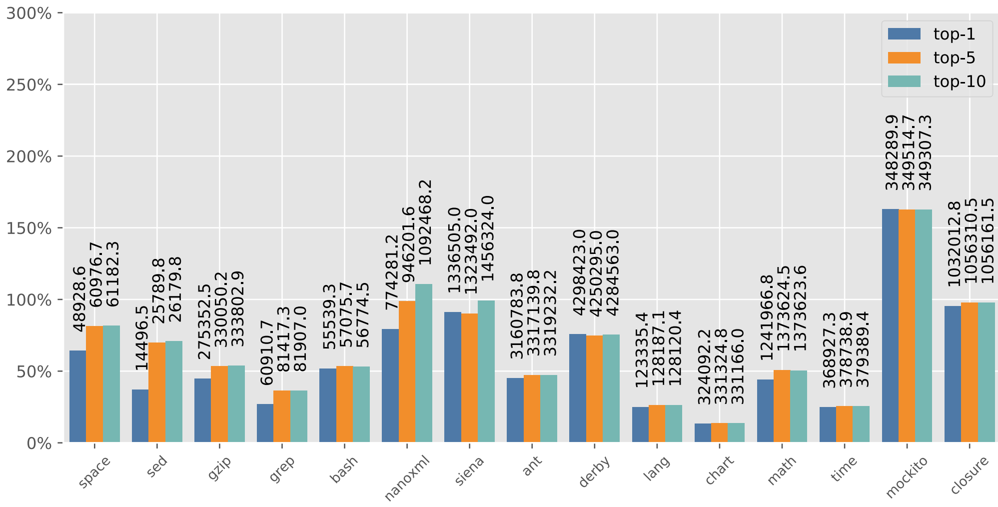
more than 90% of predicates are safely pruned away. We believe that the larger the program, the higher percentage of predicates our approach can prune away. Moreover, for different k values, the percentage varies. The smaller k is, the more predicates are pruned. This is reasonable, since a smaller k value results in a larger threshold for pruning.

We have discussed the effectiveness of our approach in pruning unnecessary predicates in Figure 6. This provides empirical evidence that ARMPS incurs relatively less time in executing instrumented programs and also utilizes less disk storage space for profiles than MPS. We further validate this hypothesis by directly measuring the execution time and the storage space for profiles.

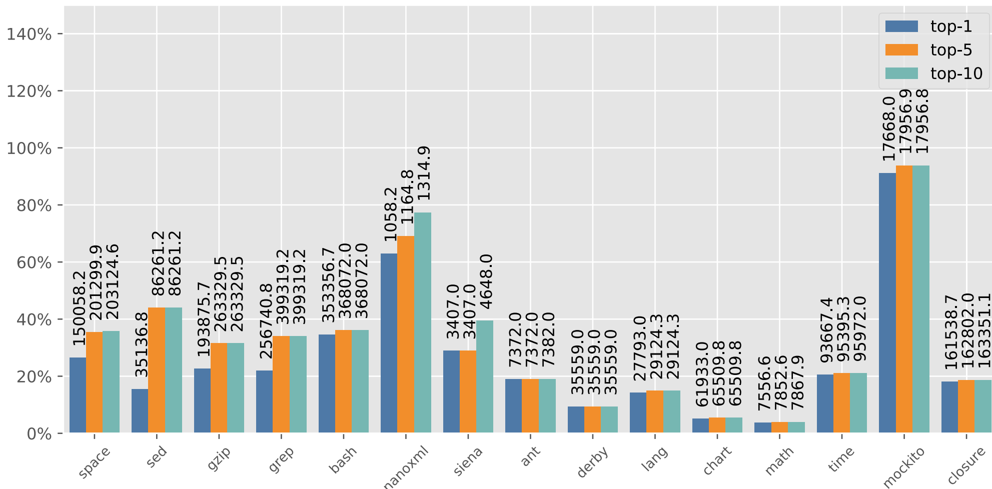
Figure 7 illustrates the percentage of execution time (Figure 7(a)) and disk space (Figure 7(b)) used for profile collection of ARMPS compared with the baseline (100%) in MPS [61]. The above costs of ARMPS involve the abstraction phase and two refinement (boosting and pruning) passes. As can be seen, on average, ARMPS takes about 63.8% of the execution time and only about 29.9% of disk space that MPS takes. For larger programs, such as *chart* and *lang*, around 62.3% and 86.9% are saved in terms of execution time and disk space, respectively. Note that although our approach ARMPS needs to run the subjects three times (one at abstract phase and two at refinement phase), the total execution time of ARMPS is still much less than that of MPS. This is because the full instrumentation of MPS could significantly slow down the original execution by several times or even more (refer to Table 4 for the overheads). The only exception is *mockito*, whose execution time is larger than that of MPS. After a deeper investigation into this subject, we found that almost all the functions in *mockito* contain only one instrumented site. As a result, the time cost of abstract phase (where only functions are instrumented) is nearly equal to the cost of MPS, which instruments all the predicates. Since our two-phase approach needs additional runs for refinement, the total execution time is larger than that of MPS.

Note that when accessing the execution time, we run each instrumented program four times, ignore the first run, and compute the average execution time of the other three runs, thus ensuring the credible and stable results.

5.2.3 Preprocessing & Mining. Having fine-grained profiles, we perform preprocessing to construct the mining dataset (Lines 8 and 14) and then mine bug signatures (Lines 9 and 15). Here, we



(a) Execution time in milliseconds

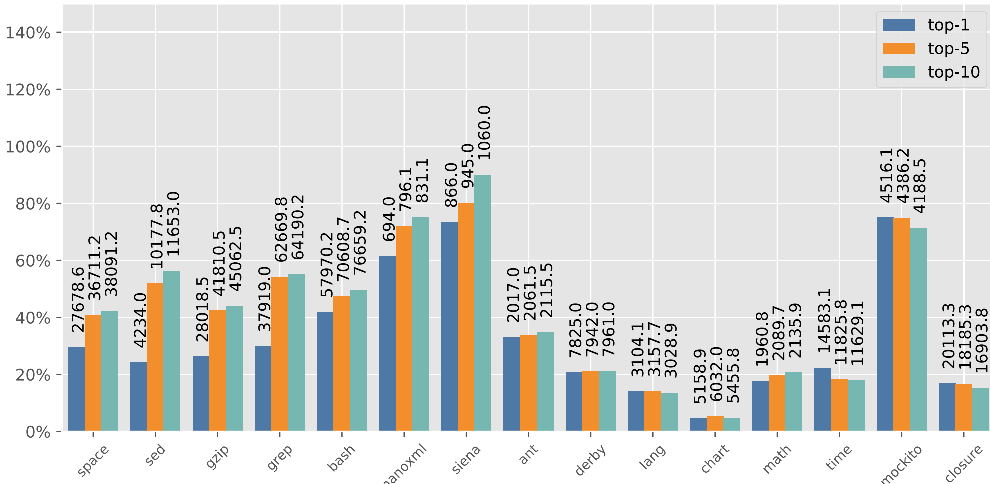


(b) Disk space in bytes

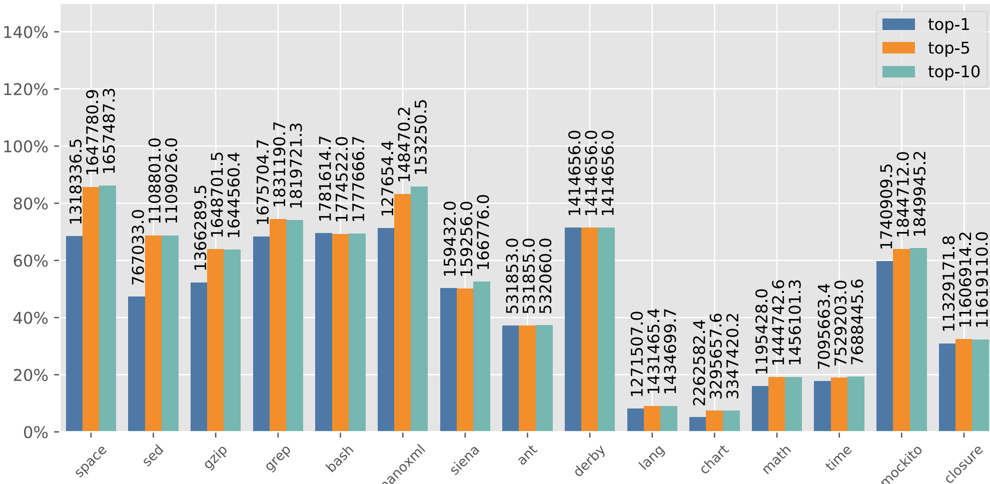
Fig. 7. Percentage and detailed amount of execution time (in milliseconds) and disk space (in bytes) used for profile collection compared with MPS [61] (100% as the baseline): (a) execution time and (b) disk space. The y-axis indicates the percentage, while the exact execution time and amount of disk space used with one decimal are shown on top of each bar in (a) and (b), respectively.

compare ARMPS with MPS in terms of time cost and memory consumption for preprocessing and mining. Note that in our experiment, both ARMPS and MPS perform the same procedure as stated in Section 2.3 under the same setting.

Figure 8 plots the percentage of time (Figure 8(a)) and peak memory (Figure 8(b)) used for preprocessing compared against that of MPS as the base (100%). Same as the above, we show the results for different k (i.e., 1, 5, and 10). For preprocessing, ARMPS only takes at most around 32.7% of time; and the peak memory consumed is no more than 44.9% of that MPS takes for all the



(a) Time in milliseconds

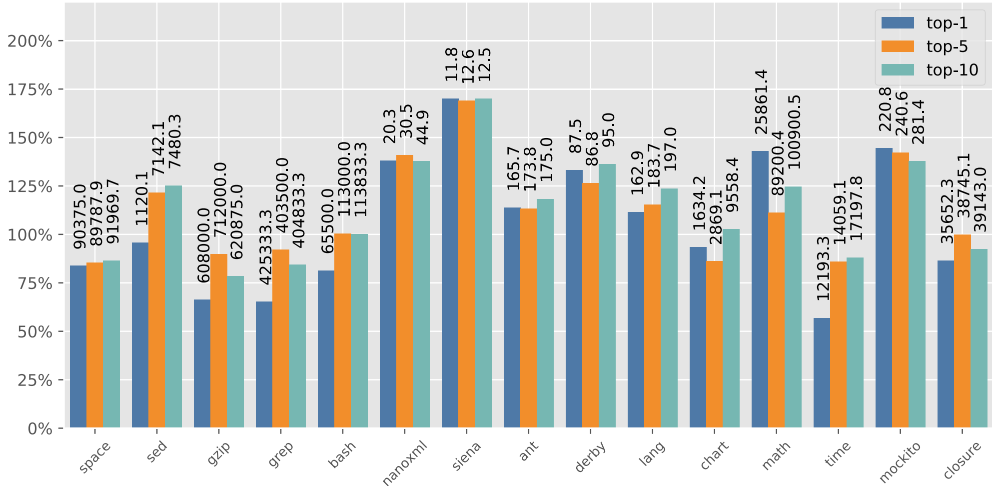


(b) Memory consumption in bytes

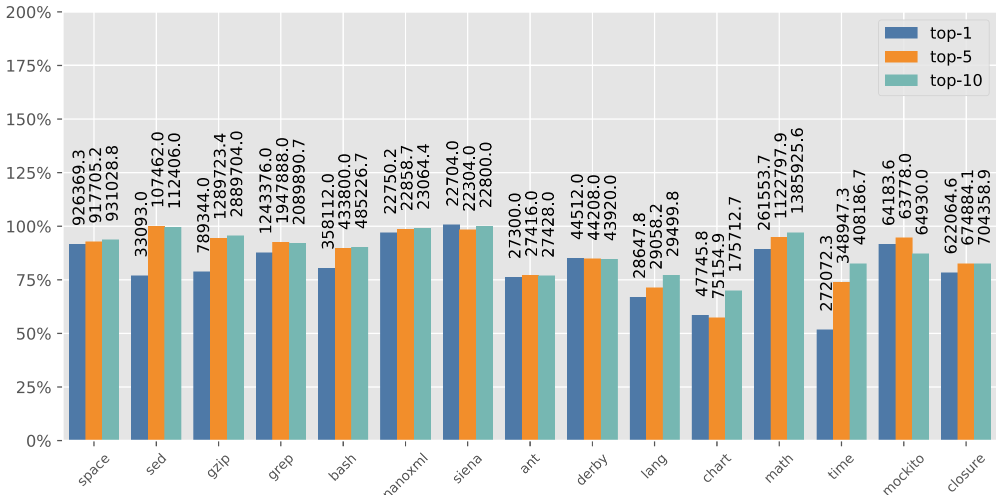
Fig. 8. Percentage and detailed amount of time (in milliseconds) and memory consumption (in bytes) for preprocessing compared with the baseline (100%) in MPS [61]: (a) time and (b) memory consumption. The y-axis indicates the percentage, while the exact execution time and amount of memory consumed with one decimal are shown on top of each bar in (a) and (b), respectively.

subjects when k is 1. On average, even when k is 10, the savings in time cost and peak memory consumption are also larger than 58.8% and 49.3%, respectively.

As for mining, the percentages of time cost and memory usage are demonstrated as Figure 9(a) and 9(b), respectively. ARMPS can save about 14.6% of peak memory consumption in general. In terms of mining time, ARMPS shows better performance on expensive mining workloads, such as *gzip* and *grep*, whose mining time is larger than 100 seconds. For *space*, *bash*, *time*, and *closure*, whose mining time is greater than 10 seconds, ARMPS also reduces the mining cost to some extent. For other subjects including *sed*, *nanoxml*, *siena*, *ant*, *derby*, *lang*, *chart*, and *mockito*, the total



(a) Time in milliseconds



(b) Memory consumption in bytes

Fig. 9. Percentage and detailed amount of time (in milliseconds) and memory consumption (in bytes) for mining compared with the baseline (100%) in MPS [61]: (a) time and (b) memory consumption. The y-axis indicates the percentage, while the exact execution time and amount of memory consumed with one decimal are shown on top of each bar in (a) and (b), respectively.

mining time of ARMPS is larger than that of MPS. This is because their workloads are small, the total time is thus dominated by the multiple invocations of mining process in ARMPS.

Generally, the savings in both time and memory consumption for mining are less than that for preprocessing. The underlying reason is that a certain number of predicates have been pruned through the filtering strategy during preprocessing discussed in Section 2.3, which weakens the pruning effectiveness of our technique. This is especially the case for subject *math*, where the preprocessing filters nearly all the needed predicates. Since our approach ARMPS needs one more mining step, the time cost is around 1.3× as large as that of the baseline.

ALGORITHM 3: Cooperative Bug Isolation with Abstraction Refinement (ARBI)

```
Input: buggy program G, number of predicates returned k
Output: top k suspicious predicates P_k
/* abstraction phase: coarse-grained instrumentation and analysis */
1  Instrument all function entries in the entire program G;
2  Deploy the instrumented program;
3  Collect sufficient function profiles;
4  C ← compute the abstract suspiciousness metric using the Ochiai metric;
5  Φ ← build the abstract suspiciousness measure using C on the collected abstract profiles;
/* refinement phase: fine-grained instrumentation and analysis */
6  Ψ ← ∅;
7  θ ← 0;
8  while Φ ≠ ∅ do
9      m ← function with the highest C value from Φ;
10     Φ ← Φ − m;
11     if C(m) < θ then break;
12     Instrument all predicates in function m;
13     Re-deploy the instrumented program;
14     Collect sufficient fine-grained profiles;
15     Update the concrete suspiciousness measure Ψ;
16     θ ← the top kth Ochiai value of predicate in Ψ;
17 end
18 return Ψ;
```

6 APPLICATION 2: PRODUCTION-RUN STATISTICAL DEBUGGING

In this section, we illustrate the employment of our technique to production-run debugging, in particular, **cooperative bug isolation (CBI)** for field failures [38]. Different from in-house debugging where developers run the instrumented program to obtain execution data, CBI directly gathers execution profiles from end-users running the deployed instrumented program. Therefore, besides considering the scale of execution data collected and analyzed by developers, we have to consider another important performance aspect—user's overhead for running the instrumented programs. To this end, we conduct an iterative refinement phase for production-run debugging so only one function is refined at each iteration.

6.1 Design and Implementation

Based on the abstraction refinement-based technique, we propose a cooperative **bug isolation approach with abstraction refinement (ARBI)**. Briefly, the workflow of ARBI is as follows:

(1) Instrument function entries, deploy the instrumented program, and collect function profiles;
(2) Compute C with the *Ochiai* metric together with the numbers of passing and failing runs P and N using Algorithm 1;
(3) Compute the abstract suspiciousness measure Φ using C on the collected function profiles;
(4) Iteratively refine functions from Φ to build the concrete suspiciousness measure Ψ until termination.

Algorithm 3 depicts the pseudo code of ARBI, which consists of an abstraction phase followed by an iterative refinement phase. At the abstraction phase, a function-level instrumentation is

performed to instrument all the function entries of the entire program, and then the instrumented program is deployed in the field (Lines 1–2). We collect sufficient execution data from a large number of end-users running the instrumented program (Line 3). Given the suspiciousness metric *Ochiai* and total numbers of failing and passing runs collected P and N, we compute the abstract metric C according to Algorithm 1 (Line 4). Having C, we analyze the collected execution data (i.e., abstract profiles) and eventually obtain the abstract suspiciousness measure Φ, i.e., a list of functions reversely ordered by their respective C values (Line 5). This list provides the abstract execution information that will be utilized to guide the iterative refinement.

At each iteration of refinement phase, we instrument at most one function at fine granularity. We select the function m with the highest C value in Φ (Line 9). Based on the feedback θ, which is the top-kth *Ochiai* value in Ψ explored so far, we check the necessity of refining function m by comparing its C value against θ. If $C(m) < \theta$, then we can safely terminate the refinement, since all the remaining functions in Φ do not contain any predicate whose *Ochiai* value can be greater than or equal to θ. Otherwise, we instrument all predicates within m according to the instrumentation scheme of CBI discussed in Section 2.1 and redeploy the instrumented program (Lines 12–13). We wait for sufficient execution data to be collected (Line 14). We next compute the *Ochiai* value of predicates in m and update the top-k predicates Ψ as well as the top kth *Ochiai* value θ (Lines 15–16). If the current list of functions is not empty, then we proceed to the next iteration. It is worth noticing that each function in the list can be examined at most once as it is removed from the list after it is considered (Line 10). The following gives more details about the re-deployment of instrumented program (Line 13) and profile collection (Lines 3 and 14):

Re-deployment. No user involvement is needed in any of these steps: Program (re-) instrumentation and (re-)deployment is completely automated. To achieve efficiency in deployment, we do *not* re-instrument or re-deploy the *whole* application in each iteration. We only recompile the changed component, which is essentially one function selected in each iteration. The component recompiled and redeployed at each iteration is often small in size (demonstrated empirically in Table 5).

For a C program, the changed component (i.e., one source file and/or header file) is re-compiled as a patch to a shared library; while for Java, a new class file is generated. Version update is done by utilizing a function wrapping mechanism—we wrap a function in a special way so calls to this function can be intercepted and rerouted to a specific instrumented version of the function in the library.

Note that re-deployment could be a problem for certain systems that consider dynamic instrumentation as a security risk. For now, we restart all re-instrumented programs after re-deployment. **Dynamic software update (DSU)** techniques [46, 60] can be employed in the future to avoid restarting the program.

Profiling Sufficiency. Like all statistical approaches, statistical debugging relies on a large amount of data to ensure the stability of results. In other words, given a sufficiently large number of execution profiles, the same results (e.g., the same ranking of functions and predictors) can be obtained even if the individual profiles under analysis are different. In our implementation, we need sufficient profiles from both the abstraction phase (Line 3) and each refinement step in the iterative phase (Line 14).

To achieve the goal, we apply an iterative, fixed point-based profile collection strategy, shown in Figure 10. Specifically, we start off by collecting a small number of profiles, on which our analysis is run to generate debugging results. Collection of additional profiles depends on whether the results are stable enough. The arrival of new profiles triggers the re-analysis of all (old and new) profiles, and the process is repeated until a fixed point is reached—no difference can be found in the results

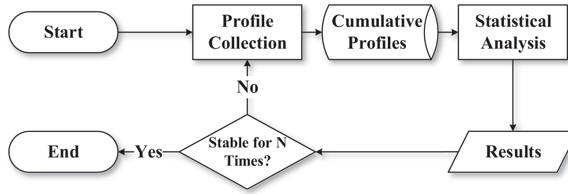


Fig. 10. Profile collection strategy.

(i.e., the ranking of functions and predicates) generated in the last N iterations (e.g., $N = 3$ is used in our experiments).

Distributed Environment. We have created an additional distributed program for our Java-based implementation that runs on a master and a set of slave nodes. The master program simulates the developer who performs the debugging while slave programs simulate users that run the instrumented program. The master instruments the program in each iteration and sends the recompiled class files to the slaves. The instrumentation code records traces at each slave and sends them back to the master over the network. The master then analyzes them and performs refinement. The master-slave communication is done over the socket. Having a distributed implementation enables us to measure performance factors such as profile sizes and network traffic, which have never been evaluated in existing statistical debugging techniques.

6.2 Experimental Evaluation

We adopted the same subject programs as that used in Section 5, whose details are shown as Table 2. Similarly, the three types of predicates introduced in Section 2.1 are taken into consideration. Same as Section 5.2, our evaluation focuses on understanding various performance factors, including, for example, instrumentation effort, user-side runtime overhead, and network traffic and latency.

To have a thorough assessment of these factors, we have designed two sets of experiments, conducted in two different environments. First, we experimented with both our C and Java implementations on a single commodity desktop. This set of experiments focuses on evaluating the instrumentation effort and the user-side runtime overhead incurred by instrumentation code. Second, we ran our distributed implementation on a cluster that simulates real-world usage of statistical debugging of Java programs in SIR repository. The goal is to understand important performance factors such as network traffic and latency; these factors are impossible to measure on a single machine.

6.2.1 Results on Single PC. All the experiments done in this subsection were run on a machine that has an Intel Core i5 3.30 GHz CPU and 16 GB main memory, running 64-bit Ubuntu 16.04. Through these experiments, we would like to answer the following four research questions:

- Q1: Compared to the traditional CBI approach where all predicates are instrumented and tracked, how many predicates does our **abstraction refinement-based bug isolation (ARBI)** approach need to instrument and track?
- Q2: How well does our approach perform for different types of predicates?
- Q3: What is the user-side overhead of our approach? How does it compare to the overhead of CBI, with and without sampling [38]?
- Q4: How does ARBI compare to **adaptive bug isolation (ABI)** [10]?

Q1: Predicates Instrumented. We used the traditional CBI as the baseline (that instruments all predicates) and ran it with the entire test suite for each program. For our technique, we

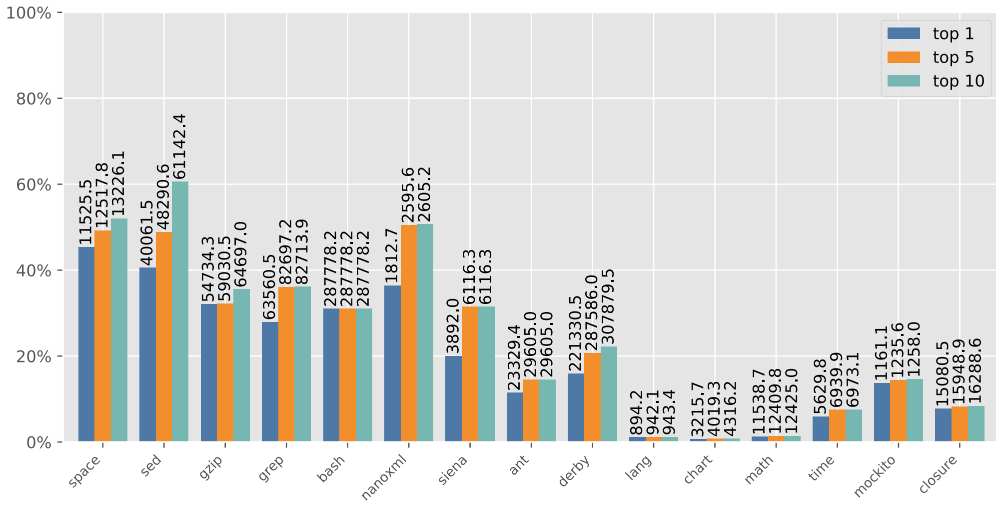


Fig. 11. Percentage and detailed number of predicates instrumented compared with 100% full instrumentation in CBI [38]. The y-axis indicates the percentage, while the number of predicates with one decimal are shown on top of each bar.

first obtained abstract suspiciousness information and then performed iterative refinements by running the same test suite for each iteration until termination. During the process, we measured the percentage of predicates instrumented to obtain the same top k predictors. We assume the test suite is large enough with sufficient failing test cases to make both CBI and ARBI reach the fixed points (i.e., statistically meaningful results).

Figure 11 shows the percentages of predicates instrumented by our approach to find the top k predictors for each program (averaged across all faulty versions), with k being 1, 5, and 10. Note that the baseline always instruments 100% of predicates. Our approach yields an overall 77.6% reduction in the number of predicates instrumented, without requiring any manual effort. We observe that our approach performs better for larger programs, especially for Defects4J subjects. In particular, for the subjects including *lang*, *chart*, *math*, *time,* and *mockito*, more than 90% of predicates were pruned away.

Q2: Effectiveness on Different Predicate Types. We ran ARBI over the entire test suite by instrumenting one of the three types of predicates (branches, returns, scalar-pairs) at each time. We compared the numbers of refinement steps needed by each type of predicate to reach termination.

Table 3 shows, for each program, the average number of iterations required (Column "Iter") to find the top five predicates and the average percentage of predicates instrumented ("Pred(%)") across multiple faulty versions of the program.

Clearly, our approach can effectively prune the predicate space for all the three types. Nevertheless, the effectiveness of our approach for different types of predicates does not differ much. In some large programs, such as `ant` and `derby`, the number of iterations needed for scalar-pair predicates is much smaller than that for the other types. This is potentially because scalar-pair predicates are much more dense than the other types of predicates. If such a predicate is highly correlated with a bug, then it can quickly distinguish itself by gaining high suspiciousness.

Q3: User-side Overhead. To answer this question, we compared the running time of each program instrumented under ARBI and under CBI with three different sampling rates: 1/1, 1/100, and 1/10, 000; 1/1 means no sampling.

Table 3. Average Numbers of Iterations Needed for Each Type of Predicate to Find the Top Five Predicates, as Well as Average Percentages of Predicates Instrumented during the Process

	Branch		Return		Scalar-pair	
Subject	Iter	Pred(%)	Iter	Pred(%)	Iter	Pred(%)
space	80.06	69.05	58.35	70.22	58.09	45.54
sed	50.75	67.72	40.00	62.84	37.25	60.49
gzip	32.69	47.20	17.23	36.75	24.54	32.24
grep	40.80	52.55	17.60	39.47	18.60	36.03
bash	239.00	27.96	225.67	23.38	238.17	31.38
nanoxml	49.09	49.23	36.18	51.33	12.95	61.16
siena	29.00	24.22	16.00	14.73	25.00	35.70
ant	490.75	21.87	385.75	20.63	190.75	12.30
derby	4,425.33	41.45	1,890.67	19.66	1,436.67	21.43
lang	6.60	0.98	5.74	1.33	4.98	1.31
chart	45.64	1.75	24.52	1.18	17.32	0.69
math	18.74	1.39	16.16	1.12	17.74	1.37
time	65.22	8.26	63.65	4.59	42.70	8.44
mockito	62.89	18.50	45.57	14.78	23.54	13.23
closure	341.98	10.58	282.53	9.43	127.45	7.34
GeoMean	–	29.51	–	24.76	–	24.58

Recall that our ARBI is iteration-based and each iteration instruments predicates in only one function. In practice, the program instrumented in different iterations would likely be executed at different user machines, and thus each user only needs to pay the overhead of heavy predicate instrumentation in one function. To assess this overhead, we measured the *average* execution time of the instrumented program across iterations (phase 2), as well as the time spent on phase 1 for executing the program with only function entries instrumented.

To precisely measure running time, we ran each faulty version of each program four times and took the average of the execution times of the last three runs. Our results are shown in Table 4. The overhead is measured as the ratio between the execution time of the instrumented run and original run without any instrumentation. Phase 1 and phase 2 under "ARBI" report, respectively, the overhead of the function-entry instrumentation in phase 1 and the average overhead of the predicate instrumentation in phase 2 across iterations. The numbers reported in this table may look different from those reported in References [38] and [59] for several common programs used, because we considered three types of predicates in our evaluation, while References [38] and [59] evaluated performance only with branch predicates instrumented.

Overall, our iterative refinement (Column "Phase 2") suffers a much lower overhead than sampling even with the lowest 1/10,000 rate. This is especially the case for C programs, because C programs usually have a relatively smaller number of functions than Java programs, and thus the number of predicates within each function in C is often much larger than that of Java, as shown as Table 2. For Java programs such as *derby*, *lang*, and *math*, the overhead of Phase 1 is higher than that of Phase 2, but still lower than that of CBI with 1/10,000 sampling rate. However, for other subjects such as *nanoxml*, *siena*, *mockito*, and *closure*, the overhead of Phase 1 is even

Table 4. User-side Slowdown in Times (×)

	CBI			ARBI	
Subject	1/1	1/100	1/10,000	Phase 1	Phase 2
space	1.78	1.76	1.69	1.13	1.07
sed	4.89	3.76	3.53	1.22	1.43
gzip	3.56	2.49	2.33	1.05	1.05
grep	11.82	8.66	7.92	1.14	1.31
bash	1.31	1.28	1.25	1.01	1.01
nanoxml	1.63	1.45	1.43	2.23	1.21
siena	1.02	1.26	1.10	1.41	1.001
ant	6.06	1.28	0.69	1.03	1.004
derby	5.73	2.09	1.60	1.13	1.03
lang	130.12	7.78	5.58	4.69	1.44
chart	184.05	19.84	16.55	27.99	1.19
math	756.20	79.47	33.39	30.33	11.03
time	140.21	15.08	11.23	28.48	1.28
mockito	5.80	5.48	5.37	8.65	1.23
closure	5.66	3.64	3.41	7.69	1.19

higher than that of fully instrumented CBI with 1/1 sampling rate. This is because there exist a number of "empty" methods in which no branch/return/scalar-pair predicate resides.

Q4: Quality Comparison with ABI. ABI [10] prunes the predicate space using heuristics. In Reference [10], the authors proposed two ABI analyses: a forward analysis that traverses forward the **control dependence graph (CDG)** from the main entry of the program and a backward analysis that traverses backward the CDG from the buggy point. At every control branch, the suspiciousness of the related branch predicates is used to determine the path to explore. Both approaches may likely run very long if wrong guidance is given by the heuristics. Hence, ABI requires the developer to determine when to terminate, making it difficult for us to come up with a fair comparison with ARBI. To answer question Q4, we design an experiment that compares the *Importance*[12] value of the top predicate found between ABI and ARBI as their predicate instrumentation progresses. We adopt *Importance* as the metric during the comparison, since it is the default metric used by ABI.

Figure 12 plots three curves for the program `space`, one for ARBI and two for ABI. Each curve represents how the *Importance* value of the top predicate changes as more iterations are executed and more predicates get instrumented. For ABI, we consider two heuristics, `T-Test` and `Importance`, as they are the top-performing heuristics according to Reference [10]. We ran the ABI forward analysis with T-Test and its backward analysis with `Importance`. These heuristics determine which control flow path to explore at each branching point.

This figure clearly shows that ARBI can quickly reach a much higher *Importance* value (beyond 0.8), while the two ABI approaches constantly stay below 0.6. Since heuristics do not provide

[12] *Importance* is a statistical metric that takes into account the sensitivity (meaning how much the entity accounts for failing runs) and specificity (meaning how less the entity is correlated to passing runs) for a given predicate. Please refer to Reference [38] for details.

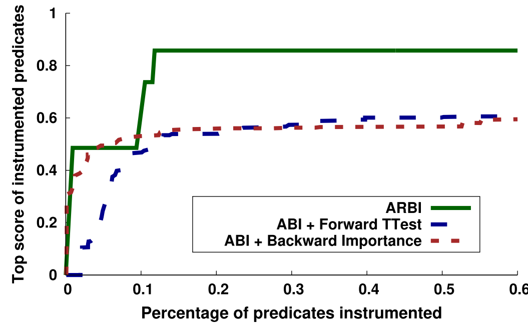


Fig. 12. Top *Importance* value comparisons for program space between ARBI and two state-of-art ABI approaches. The forward analysis uses *T-Test* as the heuristic, and the backward analysis uses *Importance* as the heuristic.

guarantees, they may likely give wrong guidance, making an ABI analysis choose to explore paths that have nothing to do with the bug.

6.2.2 Results on Distributed Environment. Our distributed environment is a local 12-node cluster, each node with two Intel Xeon E5 2.60 GHz CPUs and 32 GB RAM, running CentOS 6.6; nodes are connected by an InfiniBand network. We explore the following question:

- Q5: How much data transferred over the network can be saved by ARBI compared to CBI, and how much is the network latency?

Different from the experiments on a single PC where both CBI and ARBI ran the entire test suite for each benchmark, here, we employed the fixed-point-based strategy as discussed in Section 6.1—we kept feeding randomly selected test cases to the instrumented program in each iteration until their results became stable. This is because the goal of this set of experiments is to simulate a real-world usage scenario in which tests are run at many different users and no profiles collected are exactly the same. A technique that can quickly reach the fixed point needs to collect less profiles and thus has lower communication overhead and debugging latency. Since we would like to compare the size of data transferred over the network before reaching the fixed points between different approaches, running the whole test suite in each iteration would defeat this purpose.

To answer Q5, we measured the total size of data transferred over the network. We would also like to understand the latency of ABI. However, since ABI cannot terminate automatically, manual inspection is needed in every iteration, making it incomparable with CBI and ARBI, which are completely automated. Work from Reference [10] evaluates ABI using known bugs as "oracles." While we can also use oracles, we decided not to in the experiments, because oracles do not exist in real settings and yet the goal of this experiment is to understand ARBI's real-world impact.

Q5: Network Traffic and Latency. Table 5 compares the numbers of profiles (runs) needed and the total sizes of data transferred over the network for each Java program for CBI under the 1/100 sampling rate and ARBI. 1/100 was used to sample predicates, because much existing work [38, 59] has reported that this rate achieves a balance between the number of profiles needed (i.e., latency) and the user-side overhead. Column **Trace** reports the numbers and total sizes of profiles collected (i.e., "upstream" data that flows from slaves to the master), while Column **Binary** reports the size of "downstream" data flowing from the master to slaves. The upstream and downstream data consist primarily of traces and recompiled class files, respectively.

Compared to CBI, ARBI requires 40.3% less profiles (runs). When taking into account the size of profiles, our savings become huge—the amount of data transferred over the network by ARBI is on

Table 5. Data Transferred over the Network for CBI under the 1/100 Sampling Rate and ARBI; **Reduction** Reports ARBI's Reduction in the Total Data Size (i.e., Trace + Binary)

	Trace (#/MB)				Binary (MB)		
	CBI		ARBI		CBI	ARBI	Reduction
Subject	Num.	Size	Num.	Size	Size	Size	Total Size
nanoxml	5,640	32.5	2,520	0.8	0.3	0.4	96.3%
siena	13,602	321.3	5,773	1.9	0.3	0.4	99.3%
ant	12,516	3,144.7	10,782	5.6	6.2	6.1	99.6%
derby	9,288	15,350.4	6,078	12.1	53.2	43.9	99.6%

average 1.3% of that by CBI. This is because CBI instruments all predicates, while ARBI instruments only predicates in one function at a time. Hence, a profile collected by ARBI is orders-of-magnitude smaller than that collected by CBI. Note that we can configure ARBI to produce more amount of data per profile in exchange of fewer profiles (runs) by refining more than one function in one run, as discussed in Section 7.

We did not measure the network latency, because it would vary a lot depending on network types and configurations. We believe ARBI will have much smaller network latency than CBI in practical settings, where users and the developers are in different networks.

6.2.3 Threats to Validity. Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the performance benefit of using our two-phase statistical debugging approach on simulated environments, and thus we are unable to definitively state that our findings will hold for programs in general. For example, the single machine environment is not where the technique is intended to be used. While the distributed environment is based on multiple machines, it uses a very fast network, which a real-world deployment usually does not have.

However, we are confident that these results are indicative of the real-world impacts of our approach. First, we only used the single machine to evaluate how many predicates get instrumented and the user-side instrumentation overhead. Our results should not depend on the execution environment and thus would not change much when our approach is used in a real setting. Second, although the distributed environment uses a fast network, we reported both the total number and the total size of traces needed to complete the debugging process. These numbers would not change in a real setting. In a slower network, much more significant latency reduction should be expected, as communication and data transfer is often a bottleneck.

Another potential concern is the limited number of tests used to simulate real user inputs. The statistical soundness of a debugging technique relies on the assumption that there are sufficient failure and success runs collected from users, and they achieve sufficient code coverage. While our test suites are reasonably large and they were generated to cover different behaviors of the program, we cannot guarantee that they are sufficient. Hence, the fixed points reached in our iterative process may potentially be "local" fixed points and thus the correlation between our function/predicate ranking and their true suspiciousness may be spurious.

However, we are reasonably confident that our fixed points reflect truly stable results, because our validation that runs each debugging task for each program 100 times shows that more than 80 times ARBI and CBI produced the exact same top predictors. For the remaining times, the predictors reported were slightly different but still had large overlaps.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. Our experiments measured the costs involved

in running instrumented programs and performing the debugging process in terms of computational time and trace numbers/sizes. Although our results give an indication of the degree of such costs, our implementation can be greatly optimized in both regards. For example, our tracing information is verbose and our implementation is not optimized. However, this limitation does not affect the overall result, as this same implementation was used for both treatment techniques; i.e., the direction and magnitude of the difference between the results should not significantly change when these factors are optimized.

7 DISCUSSION

This section discusses several important issues that concern the soundness and practicality of the proposed technique.

7.1 Statistical Soundness

It is important to understand whether and under what condition the mathematical guarantees of our debugging process hold. In theory, for the abstract suspiciousness to be a sound abstraction of the concrete suspiciousness, the coarse-grained phase and each refinement must execute on exactly the same set of test cases. Therefore, in an in-house debugging scenario (Section 5) for a deterministic system, our approach is 100% mathematically sound.

However, in the cooperative debugging scenario where test cases executed at different phases may be different, our approach cannot achieve mathematical soundness. To mitigate this problem, our approach currently relies on a large number of profiles to ensure the stability of results, like many other statistical approaches, as discussed in Section 6.1. That is, we assume that, with sufficient profiles, the *Ochiai* values and hence the ranking among functions and predicates will become stable. This assumption matches with our empirical results—the diagnosis results from our approach are valid in a random distributed setting (Section 6.2.2).

A promising direction worth exploring in the future is to turn the mathematical properties stated in Section 3.2 into *statistical properties*. For example, we can incorporate confidence intervals in our metric design and turn suspiciousness scores into random variables. In this way, suspiciousness comparisons (e.g., whether the I value of a predicate is greater than a C value of a function) become hypothesis tests and their results carry statistical meanings.

7.2 Multi-level Abstractions

In this article, we only consider two levels of instrumentation in a program. However, our technique is not restricted to these two particular levels (i.e., function-level and predicate-level) and can be easily extended to consider other types of program entities, such as basic blocks, classes, and components.

For example, if a program has a very large code base and a great number of functions, then the cost and overhead at function-level instrumentation may be too high, e.g., the subjects in Defects4J whose overheads are shown in Table 4. In this case, we may consider class as a new level of abstraction. We exploit the class-level information to prune away unnecessary instrumentation of functions at the first place. Only a portion of functions in the whole program are thus instrumented at the function-level.

As another example, if a program has large functions (e.g., big C programs without a modular design), then the execution information of this function may not be a sufficiently precise approximation to the information of enclosed predicates. Moreover, the user-side overhead for running even one function with instrumented predicates may not be acceptable. In this case, we can add basic blocks as a new abstraction layer to not only improve the pruning effectiveness but also reduce the overhead. Before predicate instrumentation, we first examine and prune basic blocks. The

refinement process has a nested structure: the refinement of basic blocks would be nested within the refinement of functions. The algorithm for a general n-level abstraction refinement can be easily derived by recursively invoking the current algorithm that refines on a two-level abstraction hierarchy.

7.3 Multi-version Deployment & Multi-function Instrumentation

For production-run debugging (discussed in Section 6), our technique as an iterative approach needs to run the multi-iteration refinement process. Like all sampling-based debugging or iterative debugging techniques [6, 10, 29, 37], our technique reduces the user-side overhead of running a heavily instrumented program, but increases the latency of collecting debugging information. This tradeoff is widely considered to be worthwhile by previous work, considering the stringent user-side overhead requirement. Since there are a large number of end-users, we can simultaneously deploy programs with different instrumentations to different users for execution. As a result, the iterative process can be "parallelized," thereby reducing the waiting time for profiles. Specifically, we can remove top n functions from Φ at a time, refine them to obtain n versions of instrumented programs—each version with one function refined—and deploy them to different user sites. Consequently, developers can quickly collect execution profiles.

The number of functions to be refined in each iteration essentially defines a tradeoff framework—more functions-per-iteration means fewer iterations, but more overhead, less predicate pruning, and more total debugging cost. Under the tradeoff framework, one extreme corresponds to the two-pass design adopted by the in-house debugging (discussed in Section 5) where all the prospective functions are refined within one iteration (i.e., the pruning pass). Another extreme is to expand one function at a time (same as what we discussed in Section 6), because reducing user-side overhead is the primary goal. In fact, we can flexibly tune the tradeoff and refine multiple functions (i.e., instrumenting predicates in multiple functions) at each iteration to reduce the number of iterations in production-run debugging.

7.4 Applicability

Our technique works for all types of predicates and all types of statistical metrics, as long as the statistical debugging approaches only distinguish whether a predicate has been observed at least once or never in a run. In other words, our technique would fail if debugging relies on the *exact* number of times a predicate is observed in each run—no mathematical relation can be established between the parameters of I and C, and thus C is no longer valid.

Besides statistical debugging, our technique also has the potential to be applied to a wider class of program analysis tasks that requires code instrumentation, e.g., time-travel debugging, grey-box fuzzing, and automated program repair.

8 RELATED WORK

While there exists a large body of existing work, this section focuses on the discussion of work that is most closely related to our technique.

8.1 Statistical Debugging

Statistical debugging[13] [53, 66, 74] is a family of approaches that isolate a single suspicious element as the root cause of failure by identifying the discriminative behavior between passing and failing execution profiles. There are two essential issues to consider in statistical debugging:

[13]Some also call it spectrum-based fault localization or statistical fault localization. We treat them equally in this article.

type of program entity capturing program execution behavior and discriminative metric utilized to quantify the suspiciousness of program entities.

Various types of program entities have been adopted to capture the execution information, e.g., the statement coverage [1, 33], basic blocks [15] or functions [20] executed, def-use pairs [58], data predicates [38, 40] tracked, and interleaving patterns [29]. Meanwhile, a great variety of statistical discriminative metrics are adopted to quantify the correlation between program entities and bug causes [4, 33, 41, 45, 64].

Tarantula [33, 34] and Ochiai [1, 4] both use statement coverage information to represent the execution behavior and assess the suspiciousness of each statement based on their proposed suspiciousness measures. Nainar et al. [9] introduced *complex predicates* composed of atomic predicates from Reference [38] using logical operators (such as conjunction and disjunction). Gore et al. [23] also extended Reference [38] by introducing the notion of *elastic* predicates such that statistical bug isolation can be tailored to a specific class of software involving floating-point computations and continuous stochastic distributions. Furthermore, several researchers applied causal inference to reduce the control and data flow dependence confounding biases [44, 52] in statement-level [12, 13] and also failure flow confounding bias in predicate-level [24] statistical bug isolation. Recently, Jiang et al. [27] and Zou et al. [74] studied the combinations of multiple approaches and proposed the unified model, which significantly outperforms any individual technique.

To better support debugging, more information than sole buggy statement or root cause is required. Hsu et al. [26] coined the term *bug signature*, which comprises multiple elements providing bug context information, and adopted sequence mining algorithm to discover longest sequences in a set of failing executions as bug signatures. Cheng et al. [15] mined the discriminative control flow graph patterns from both passing and failing executions as bug signatures. To enhance the predictive power of bug signatures, Sun and Khoo [61] proposed *predicated bug signature mining*, where both data predicates and control flow information are utilized. In addition, Jiang and Su [28] proposed context-aware statistical debugging by first identifying suspicious bug predictors via feature selection, then grouping correlated predictors, and finally building faulty control flow paths linking predictors to provide contextual information for debugging.

The problem with these statistical debugging approaches (statistical bug isolation [1, 4, 23, 33, 34, 38] and bug signature mining [15, 61]) is that they consider *every program statement* to be potentially relevant to the bug, and thus instrument the entire program to obtain the full-scale execution information for debugging. In fact, most program code works well. Such full-scale instrumentation costs dearly in terms of wastage of execution time, storage space, CPU and memory usage. This article presents a selective instrumentation technique where only the suspicious program elements are instrumented, ignoring irrelevant ones, thus achieving more efficient debugging. Our technique is general enough so it is applicable to most of the above statistical debugging approaches that satisfy the requirements discussed in Section 7.4.

8.2 Cooperative Bug Isolation for Field Failures

Most software deployed remains buggy in spite of extensive in-house testing and debugging. These failures that occur after deployment on user machines are called *field failures* [18, 30]. To debug these field failures, developers are required to reproduce them in the lab according to the bug reports and then perform the same process as in-house debugging. However, in practice, it is hard for developers to reproduce field failures (especially client-side failures) in the lab due to the different environments, configurations, and/or nondeterminism. In References [38, 39], Liblit et al. proposed a different execution data collection scheme for debugging field failures. They deployed the instrumented program to users and collected user's execution profiles for debugging with user's approval. They term this approach *cooperative bug isolation* [39]. To ensure the sufficiently

low runtime overhead, sparse random sampling [5, 37] is adopted where only a sparse and random subset of predicates are sampled to record their execution information while the instrumented program is running. This sampling technique amortizes the monitoring cost to a considerable number of end-users so each user suffers a relatively low time overhead. Nevertheless, from the perspective of developers, the total monitoring overhead and the total size of execution data collected and analyzed remain unchanged. In this sense, the cooperative approach still suffers the same problem as mentioned earlier that: Plenty of unnecessary execution data is collected, and it consumes many resources such as network bandwidth, storage space, CPU time, and so on.

As an extension of cooperative bug isolation, *iterative (cooperative) bug isolation* approaches [10, 16] have been proposed to ensure minimal effort spent by both end-users and developers. These approaches perform the instrumentation and statistical analysis in an iterative manner. Specifically, Chilimbi et al. [16] monitored a set of functions, branches, and paths to analyze whether these are strong predictors of the failure at each iteration. If so, then they terminated the iterative process by returning these strong predictors. Otherwise, they expanded the search via a static analysis to monitor other parts of code closely interacting with the weak predictors. Similarly, based on the locality principle, Arumuga et al. [10] proposed an adaptive monitoring strategy that monitors a few predicates at each iteration and adaptively adjusts the instrumentation plan to include predicates close to the highly suspicious predicate currently explored.

However, there are two main drawbacks of iterative approaches. First, both iterative approaches [10, 16] make use of the principle of locality to guide their search for bugs; this principle, however, is not always effective in localizing bugs, as experiments have found [10]. Second, both iterative approaches require developers to manually check the predictors reported at each iteration until the bug cause is found. As claimed in Reference [51], developers are reluctant to go through a list of predictors, not to mention the need to repetitively perform this check at every iteration. Here, we proposed an automated predicate-space pruning technique for statistical debugging, which can greatly reduce the number of predicates that need to be profiled and analyzed, while never missing top-ranked failure predictors with mathematical guarantees.

8.3 Other Automated Debugging Approaches

Program slicing [42, 62, 72] is a commonly used technique for debugging. Zeller et al. proposed *delta debugging* to isolate the failure-inducing difference in source code [68], inputs [69], and program states [19] between one failing and one passing run. Techniques have been proposed to improve delta debugging by combining it with dynamic slicing [25] and by better selecting the pair of passing run and failing run [55, 56]. Similarly to delta debugging for program states, Zhang et al. [71] forcibly switched the branch predicate's outcome in a failing run and localize the bug by examining the predicates whose switching produces correct result.

Mutation-based fault localization [43, 48] exploits the information of mutation analysis and computes the suspiciousness score for each statement. If the mutation to a statement affects the failing runs frequently, while the passing runs rarely, then it is potentially suspicious.

Information retrieval-based debugging approaches have been recently studied [57, 63, 70, 73]. Instead of analyzing the runtime execution trace, these approaches take a bug report as input and apply the information retrieval techniques to pinpoint a list of relevant source files to the bug. There are also studies [11, 14, 65, 67] that leverage machine learning techniques to locate bugs effectively.

Moreover, several debugging techniques focus on finding concurrency bugs in multi-threaded programs. Falcon [49] and its follow-up work Unicorn [50] are pattern-based dynamic analysis techniques for fault localization in concurrent programs. They combine pattern identification with statistical suspiciousness ranking of memory access patterns.

9 CONCLUSION

We devise a novel, systematic abstraction refinement-based technique performing selective instrumentation to enhance the efficiency of statistical debugging. At the abstract level, functions are instrumented to capture abstract suspiciousness information, which will be then exploited to guide the refinement of predicate instrumentation. Different from the existing approaches, our technique provides a mathematically rigorous guarantee of debugging effectiveness without the need of any developers' intervention. We apply our technique to two statistical debugging scenarios: in-house and production-run debugging. The results indicate that our technique significantly enhances the efficiency of statistical debugging, making a step forward to painless debugging.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-based multiple fault localization. In *ASE*. 88–99.
[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* 82, 11 (2009), 1780–1792.
[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *PRDC*. IEEE, 39–46.
[4] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*. IEEE Computer Society, 89–98.
[5] Matthew Arnold and Barbara G. Ryder. 2001. A framework for reducing the cost of instrumented code. In *PLDI*. 168–179.
[6] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*. 101–112.
[7] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *ASPLOS*. 207–222.
[8] Piramanayagam Arumuga Nainar. 2012. *Applications of Static Analysis and Program Structure in Statistical Debugging*. Ph.D. Dissertation. University of Wisconsin – Madison.
[9] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. 2007. Statistical debugging using compound Boolean predicates. In *ISSTA*. 5–15.
[10] Piramanayagam Arumuga Nainar and Ben Liblit. 2010. Adaptive bug isolation. In *ICSE*. 255–264.
[11] L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio. 2009. Exploring machine learning techniques for fault localization. In *LATW*. 1–6. DOI: https://doi.org/10.1109/LATW.2009.4813783
[12] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal inference for statistical fault localization. In *ISSTA*. 73–84.
[13] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2011. Mitigating the confounding effects of program dependences for effective fault localization. In *FSE*. 146–156.
[14] L. C. Briand, Y. Labiche, and X. Liu. 2007. Using machine learning to support debugging with tarantula. In *ISSRE*. 137–146. DOI: https://doi.org/10.1109/ISSRE.2007.31
[15] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. 2009. Identifying bug signatures using discriminative graph mining. In *ISSTA*. 141–152.
[16] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE*. 34–44.
[17] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *CAV*. 154–169.
[18] James Clause and Alessandro Orso. 2007. A technique for enabling and supporting debugging of field failures. In *ICSE*. 261–270.
[19] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *ICSE*. 342–351.
[20] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for Java. In *ECOOP*. 528–550.
[21] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.: Int. J.* 10, 4 (2005), 405–435.
[22] L. Gazzola, L. Mariani, F. Pastore, and M. Pezzè. 2017. An exploratory study of field failures. In *ISSRE*. 67–77. DOI: https://doi.org/10.1109/ISSRE.2017.10
[23] Ross Gore, Paul F. Reynolds, and David Kamensky. 2011. Statistical debugging with elastic predicates. In *ASE*. 492–495.

[24] Ross Gore and Paul F. Reynolds, Jr. 2012. Reducing confounding bias in predicate-level statistical debugging metrics. In *ICSE*. 463–473.
[25] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *ASE*. 263–272.
[26] Hwa-You Hsu, J. A. Jones, and A. Orso. 2008. Rapid: Identifying bug signatures to support debugging activities. In *ASE*. 439–442.
[27] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang. 2019. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *ASE*. 502–514. DOI: https://doi.org/10.1109/ASE.2019.00054
[28] Lingxiao Jiang and Zhendong Su. 2007. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE*. 184–193.
[29] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*. 241–255.
[30] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *ICSE*. 474–484.
[31] Wei Jin and Alessandro Orso. 2013. F3: Fault localization for field failures. In *ISSTA*. Association for Computing Machinery, New York, NY, 213–223. DOI: https://doi.org/10.1145/2483760.2483763
[32] James A. Jones, James F. Bowring, and Mary Jean Harrold. 2007. Debugging in parallel. In *ISSTA*. 16–26.
[33] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*. 273–282.
[34] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. 467–477.
[35] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. Association for Computing Machinery, New York, NY, 437–440. DOI: https://doi.org/10.1145/2610384.2628055
[36] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *ISSTA*. Association for Computing Machinery, New York, NY, 165–176. DOI: https://doi.org/10.1145/2931037.2931051
[37] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug isolation via remote program sampling. In *PLDI*. 141–154.
[38] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *PLDI*. 15–26.
[39] Benjamin Robert Liblit. 2004. *Cooperative Bug Isolation*. Ph.D. Dissertation. University of California, Berkeley.
[40] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical model-based bug localization. In *FSE*. 286–295.
[41] Lucia, David Lo, Lingxiao Jiang, and Aditya Budi. 2010. Comprehensive evaluation of association measures for fault localization. In *ICSM*. 1–10.
[42] J. R. Lyle and M. Weiser 1987. Automatic program bug location by program slicing. In *CSAE*. 877–883.
[43] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*. IEEE Computer Society, 153–162. DOI: https://doi.org/10.1109/ICST.2014.28
[44] Stephen L. Morgan and Christopher Winship. 2007. *Counterfactuals and Causal Inference: Methods and Principles for Social Research*, (1st ed.). Cambridge University Press.
[45] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 1–32.
[46] Iulian Neamtiu and Michael Hicks. 2009. Safe and timely updates to multi-threaded programs. In *PLDI*. 13–24.
[47] Akira Ochiai. 1957. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bull. Japan. Societ. Sci. Fish.* 22 (1957), 526–530.
[48] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based fault localization. *Softw. Test. Verif. Reliab.* 25, 5-7 (2015), 605–628. DOI: https://doi.org/10.1002/stvr.1509
[49] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault localization in concurrent programs. In *ICSE*. 245–254.
[50] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2014. Unicorn: A unified approach for localizing non-deadlock concurrency bugs. *Softw. Test. Verif. Reliab.* 25, 3 (2014), 167–190.
[51] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In *ISSTA*. 199–209.
[52] Judea Pearl. 2000. *Causality: Models, Reasoning, and Inference*. Cambridge University Press.
[53] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *ICSE*. IEEE Press, 609–620. DOI: https://doi.org/10.1109/ICSE.2017.62

[54] J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA.
[55] Emmanuel Renieris. 2005. *A Research Framework for Software-fault Localization Tools*. Ph.D. Dissertation. Providence, RI.
[56] Manos Renieris and Steven P. Reiss. 2003. Fault localization with nearest neighbor queries. In *ASE*. 30–39.
[57] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *ASE*. IEEE Press, 345–355. DOI: https://doi.org/10.1109/ASE.2013.6693093
[58] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *ICSE*. 56–66.
[59] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. In *OOPSLA*. 561–578.
[60] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. 2009. Dynamic software updates: A VM-centric approach. In *PLDI*. 1–12.
[61] Chengnian Sun and Siau-Cheng Khoo. 2013. Mining succinct predicated bug signatures. In *FSE*. 576–586.
[62] Mark Weiser. 1982. Programmers use slices when debugging. *Commun. ACM* 25, 7 (July 1982), 446–452.
[63] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *ASE*. Association for Computing Machinery, New York, NY, USA, 262–273. DOI: https://doi.org/10.1145/2970276.2970359
[64] W. E. Wong, V. Debroy, R. Gao, and Y. Li. 2014. The DStar method for effective software fault localization. *IEEE Trans. Reliab.* 63, 1 (2014), 290–308. DOI: https://doi.org/10.1109/TR.2013.2285319
[65] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. 2012. Effective software fault localization using an RBF neural network. *IEEE Trans. Reliab.* 61, 1 (2012), 149–169. DOI: https://doi.org/10.1109/TR.2011.2172031
[66] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Trans. Softw. Eng.* 42, 8 (Aug. 2016), 707–740. DOI: https://doi.org/10.1109/TSE.2016.2521368
[67] W. Eric Wong and Yu Qi. 2009. BP neural network-based effective fault localization. *Int. J. Softw. Eng. Knowl. Eng.* 19, 04 (2009), 573–597. DOI: https://doi.org/10.1142/S021819400900426X
[68] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? In *FSE*. 253–267.
[69] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* 28, 2 (2002), 183–200.
[70] Jie Zhang, Xiaoyin Wang, Dan Hao, Bing Xie, Lu Zhang, and Hong Mei. 2015. A survey on bug-report analysis. *Sci. China Inf. Sci.* 58, 2 (2015), 1–24. DOI: https://doi.org/10.1007/s11432-014-5241-2
[71] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *ICSE*. 272–281.
[72] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. 2005. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*. 33–42.
[73] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *ICSE*. IEEE Press, 14–24.
[74] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Trans. Softw. Eng.* (2019). DOI: https://doi.org/10.1109/TSE.2019.2892102
[75] Zhiqiang Zuo, Lu Fang, Siau-Cheng Khoo, Guoqing Xu, and Shan Lu. 2016. Low-overhead and fully automated statistical debugging with abstraction refinement. In *OOPSLA*. Association for Computing Machinery, New York, NY, 881–896. DOI: https://doi.org/10.1145/2983990.2984005
[76] Zhiqiang Zuo, Siau-Cheng Khoo, and Chengnian Sun. 2014. Efficient predicated bug signature mining via hierarchical instrumentation. In *ISSTA*. 215–224.

Received 14 January 2021; revised 6 April 2022; accepted 25 May 2022