A Systematic Study of Failure Proximity

Chao Liu, Member, IEEE, Xiangyu Zhang, Member, IEEE, and Jiawei Han, Senior Member, IEEE

Abstract—Software end users are the best testers, who keep revealing bugs in software that has undergone rigorous in-house testing. In order to leverage their testing efforts, failure reporting components have been widely deployed in released software. The Microsoft Dr. Watson System [1] and the Mozilla Quality Feedback Agent [2] are the two most typical examples. Many utilities of the collected failure data depend on an effective failure indexing technique, which, in the optimal case, would index all failures caused by the same bug together. Unfortunately, the problem of failure proximity, which underpins the effectiveness of an indexing technique, has not been systematically studied. This paper presents the first systematic study of failure proximity. A failure proximity consists of two components: a fingerprinting function that extracts signatures from failures and a distance function that calculates (from the extracted signatures) the likelihood of two failures being due to the same bug. By considering different instantiations of the two functions, we study an array of six failure proximities (two of them are new) in this paper. These proximities range from the simplest approach which checks failure points to the most sophisticated approach which utilizes fault localization algorithms to extract failure signatures. Besides presenting technical details of each proximity, we also study the properties of each proximity and trade-offs between proximities. Altogether these deliver a systematic view of failure proximity. For fair comparison, this study proposes the first set of evaluation metrics that objectively quantifies the effectiveness of different failure proximities. We carry out three case studies of the six proximities on three mid-sized programs (namely, flex, grep, and gzip) and evaluate their effectiveness using the proposed metrics. The experimental result clearly validates our identified properties and trade-offs. In summary, this study not only presents a systematic study of six failure proximities, the problem formulation, the proposed metrics, and the experimental result, but would also help guide further investigation in the future.

Index Terms—Failure analysis, failure indexing, failure proximity, fault localization, debugging aids, statistical debugging, software maintenance.

1 INTRODUCTION

Failure reporting is a widely deployed component in modern software and the Microsoft Dr. Watson System [1] and the Mozilla Quality Feedback Agent [2] are the two most representative examples. When a program fails, the failure reporting component automatically collects failure-relevant information, prepares a failure report, and (with the user’s permission) sends it to the software vendor for diagnosis. Nowadays, by utilizing released libraries, either that of a third party (e.g., BugSentry [3]) or Microsoft’s, any programs, regardless of their complexities, can have their own failure reporting channels. The authors of this study have seen failure reporting implemented in, e.g., Google Toolbar, FreeCall, and BitTorrent, just to name a few.

Because reported failures reflect how the software is exercised in practice and what software faults (i.e., bugs) are actually bothering end users, an in-depth analysis of them will provide invaluable guidance to software maintenance and development. Many utilities of the reported failures rely on an effective failure indexing technique, which ideally would index all failures due to the same fault together.

Should such a technique exist, analyzing the collected failure reports would become a routine work. For instance

- **Duplicate removal.** Only one or a few reported failures need to be analyzed for all failures due to the same bug.
- **Failure prioritization.** Through failure indexing techniques, groups of reported failures that are likely due to the same fault can be identified and the group size is a good heuristic for failure prioritization.
- **Failure assignment.** Some techniques, as will be presented in this study, can automatically provide the likely fault location for each failure group based on which failures can be assigned to the appropriate developers.
- **Patch suggestion.** When an end user reports a failure, indexing techniques can help search the database of resolved failures. If there is a hit, the corresponding patch can be automatically provided to the end user.

Underpinning an effective index technique is a properly designed failure proximity, which quantifies the likelihood for any two failures being due to the same fault. The higher the likelihood, the more likely the two failures should be indexed together. Failure proximity is a hard problem because it is not easy even for human beings to tell whether two failures are due to the same fault. Fortunately, as crashing failures due to the same bug tend to crash the program at the same location, the proximity between crashing failures can be easily defined and is shown to be effective for failure indexing, e.g., see [4], [5], [6].
Unfortunately, the convenient “same failure point, same fault” property no longer holds for noncrashing failures. Differently from crashing failures, noncrashing failures manifest through erroneous outputs instead of crashes and are usually caused by semantic bugs. According to a recent study of bug characteristics [7], semantic bugs are becoming increasingly dominant because many memory bugs that incur program crashes can be effectively eliminated through type-safe languages and/or memory checking tools [8], [9]. In general, the proximity between noncrashing failures is hard to quantify because discriminative failure signatures like crashing points are no longer available.

Although the crashing point is not available, one may speculate that the failure point at which erroneous outputs are first emitted would function similarly to crashing points. Unfortunately, our experiment (Section 5) indicates that this speculated approach is not effective. The explanation is simple and intuitive: Different faults can generate different erroneous outputs, but the outputs are likely emitted through the same source code location. For example, most programs use a function similar to `flush_buf` to flush the output to external devices.

A more sophisticated approach is investigated by Podgurski et al. [10], in which failure proximities are calculated based on execution profiles. Although some encouraging results are reported, proximities based on profiles can bear two limitations. First, a significant portion of execution profiles would be the same even if these failures are due to different faults (see concrete examples in Section 5). Second, profile-based proximities do not directly support failure assignment. Failures measured as close to each other here only suggest that these failures exhibit similar profiles and developers still need to reason about the likely fault location from the similar profiles to assign failures.

Besides the above-mentioned techniques, there are other studies that can be potentially adapted as failure proximities (e.g., [11], [12]). As can be expected, researchers will continue to propose new proximities and indexing techniques due to the importance of failure analysis in general. While proposing one or more techniques helps advance the state of the art, we believe that what is more important is to answer some fundamental questions, such as:

- What is failure indexing and what is failure proximity?
- What are the existing approaches in the literature and are they effective for noncrashing failures?
- What is a good evaluation metric that can be used to compare different proximities in an objective way?

In this paper, we present the first systematic study of failure proximity, which addresses the above questions. In particular, we make the following contributions:

1. We propose the first failure indexing model, which views an indexing technique as instantiations of three functions, namely, a fingerprinting function $F$ that extracts failure signatures from executions, a distance function $D$ that calculates (from the extracted signatures) the likelihood of two failures being due to the same bug, and a clustering function $C$ that groups failures based on the calculated distances. Because clustering has been extensively studied in statistics and machine learning [13], this study focuses on the choices of $F$ and $D$, which comprise the failure proximity.

2. By considering different instantiations of $F$ and $D$, we examine an array of six failure proximities: two simple approaches motivated by related studies about crashing failures, two existing approaches, and two approaches proposed in this study. In particular, one of the proposed approaches, named SD-PROXIMITY, demonstrates that fault localization algorithms are not limited to locating software faults and they can actually serve as effective fingerprinting functions.

3. In order to promote quantitative comparison, we introduce the first set of metrics that scores the effectiveness of different proximities based on which various proximities can be objectively compared. We also propose some principles to better be observed in future metric design.

4. Finally, we carry out three case studies of the six proximities on three mid-sized programs (namely, flex, grep, and gzip) and evaluate their effectiveness using the proposed metrics. The experimental result clearly validates our identified properties of and trade-offs between different proximities. This experimental result, together with the problem formulation and the proposed metrics, would help to guide future research.

The rest of this paper is organized as follows: We describe a failure indexing framework in Section 2, which comprises the problem formulation and the failure indexing model. We then study the six proximities in Section 3. We explain the proposed evaluation metrics in Section 4 and present the experimental result in Section 5. With related work and threats to validity discussed in Section 6, Section 7 concludes this study.

### 2 A FAILURE INDEXING FRAMEWORK

In this section, we formulate the problem of failure indexing in Section 2.1, describe what comprises a failure indexing model in Section 2.2, and, finally, discuss what metrics are needed to evaluate the quality of a failure indexing result.

#### 2.1 Failure Indexing in Formulation

We say a test case $t$ fails on a program $P$ if the output is different from what is expected. If it is, the test case $t$ is a failing case and the execution is a failure execution or a failure in short. On the other hand, if the output is the same as what is expected, the test case $t$ is a passing case and the execution trace is a passing execution.

Given a set of $n$ failures $X = \{x_1, x_2, \ldots, x_n\}$ caused by $m$ faults $F = \{f_1, f_2, \ldots, f_m\}$, where neither $m$ nor $F$ are known. We assume that an oracle function $\Phi: X \rightarrow F$,  

1. In this paper, we use debugging algorithms and fault localization algorithms interchangeably to refer to algorithms that automatically identify certain source code as likely fault locations.
which is also unknown, specifies the due to relationship between X and F: A failure \( x_i \) is due to the fault \( f_k \) if and only if

\[
\Phi(x_i) = k.
\]

The fault \( f_k \) is then called the root cause of the failure \( x_i \). For clarity and simplicity, this study focuses on failures that are each caused by one and only one fault and more complicated cases are briefly discussed in Section 5.

The oracle function \( \Phi \) partitions the set of failures \( X \) into \( m \) mutually exclusive and collectively exhaustive failure classes \( \{G_k\}_{k=1}^m \) with

\[
G_k = \{x_i | \Phi(x_i) = k, \text{ for } i = 1, 2, \ldots, n\}.
\]

Let us use \( \mathcal{G} \) to denote this partition, i.e., \( \mathcal{G} = \{G_1, G_2, \ldots, G_m\} \). For a given failure \( x \), \( G_{\Phi(x)} \) is then the failure class that \( x \) belongs to and \( G_{\Phi(x)} \) contains all failures due to the same fault as \( x \).

Then, the problem of failure indexing is to obtain a partition \( \mathcal{G}' \) of the failure set \( X \) such that the discrepancies between \( \mathcal{G}' \) and \( \mathcal{G} \) are minimized, where the discrepancies can be quantified in many ways (e.g., by the Rand Index [14] or by the Jaccard Index [15]). We say an indexing technique is optimal (denoted by OP-INDEX) if and only if its obtained partition \( \mathcal{G}' \) is always identical to \( \mathcal{G} \) for any set of failures on any program. Naturally, the partition \( \mathcal{G} \) is called the optimal partition.

Without concerns about costs, one can immediately develop a trivial optimal indexing technique: Locate the root cause of each failure and partition failures based on their root causes. Apparently, this is not practical because the major motivation of failure indexing is to avoid debugging every failure. Therefore, OP-INDEX is not a practical technique but an objective that various practical techniques try to approximate.

Although the above formulation of failure indexing appears to depend on fault localization, it is worth clarifying that failure indexing and fault localization are two different research problems because they aim at different targets:

- Fault localization aims at locating the root cause in the source code.
- Failure indexing aims at finding discriminative failure features to partition failures such that failures in the same partition are likely due to the same fault.

The root cause is only one of many possible discriminative features that can be used for failure indexing and a central problem of failure indexing is to discover good but obtainable discriminative features and properly use them to partition failures.

An immediate consequence of the different targets of failure indexing and fault localization is that their effectiveness would be evaluated using different metrics. For example, dynamic slices are not good for fault localization (because they contain not only root causes but also the dependence chains from root causes to failures), but they can be quite effective for failure indexing, as shown later in this paper. Enough said about what is failure indexing, let us examine what constitutes a failure indexing technique in the following section.

### 2.2 A Failure Indexing Model

Without loss of generality, a failure indexing technique \( T \) is a triplet \( < F, D, C > \), where \( F, D, \) and \( C \) are the fingerprinting function, the distance function, and the clustering function, respectively. We now explain the three functions one by one and Table 1 lists some possible instantiations for each of the three functions.

The fingerprinting function \( F \) extracts failure signatures from program failures. It is named fingerprinting because the function \( F \) basically maps a failing execution into a compact representation (i.e., the failure signature) in the hope that the failure signature is indicative of its root cause. The first row in Table 1 lists some possible instantiations of the fingerprinting function, some of which can be applied at runtime (e.g., extracting the call stack), while others may involve offline processing (e.g., computing the dynamic slice). We will explain them in detail in Section 3.

After failure signatures are extracted, the distance function \( D \) calculates the pairwise distances between failures based on the dissimilarities between corresponding signatures. Preferably, we expect the distance function \( D \) to be a metric, meaning that the following four properties are satisfied:

1. \( D(\alpha, \beta) \geq 0 \) (nonnegativity),
2. \( D(\alpha, \beta) = 0 \) if and only if \( \alpha = \beta \) (identity),
3. \( D(\alpha, \beta) = D(\beta, \alpha) \) (symmetry), and
4. \( D(\alpha, \gamma) \leq D(\alpha, \beta) + D(\beta, \gamma) \) (triangle inequality),

where \( \alpha, \beta, \) and \( \gamma \) are the signatures of three failures.

The output of the distance function \( D \) is an \( n \times n \) proximity matrix \( M \), where \( M_{i,j} \) is the distance (i.e., dissimilarity) between failures \( x_i \) and \( x_j \). The second row in Table 1 gives some instantiations of the distance function.

The third (and the last) component of failure indexing is a clustering function, which partitions failures based on the computed proximity matrix. The problem of clustering has been extensively studied for at least four decades (e.g., see [17] for a historical review) and numerous clustering algorithms have been proposed and carefully studied, e.g., K-Means clustering, hierarchical clustering, spectral clustering, just to name a few most successful ones. But, ironically, the consensus is unfortunately that “There is no clustering technique that is universally applicable in uncovering the variety of structures present in multidimensional data sets,” as

<table>
<thead>
<tr>
<th>Components</th>
<th>Possible Instantiations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>extract stack traces, compute dynamic slices, locate likely fault locations, ( \cdots )</td>
</tr>
<tr>
<td>( D )</td>
<td>0-1 distance, Euclidean distance, Jaccard distance, Kendall’s tau distance, edit distance, ( \cdots )</td>
</tr>
<tr>
<td>( C )</td>
<td>K-Means clustering, K-medoids, Single-Link, Complete-Link, DBSCAN, Spectral clustering, ( \cdots )</td>
</tr>
</tbody>
</table>
concluded by the survey in [17]. Therefore, in this study, we will not reexamine how different clustering algorithms render different clustering results for the same proximity matrix. Readers seeking advice on choices of clustering algorithms are referred to surveys like [15], [17], [18]. Instead, this study will focus more on how to produce a good failure proximity such that the distance between failures due to the same bug is small and is large otherwise. Therefore, the problem of failure proximity concerns how to design a fingerprinting function $F$ that extracts discriminative signatures from failures and how to use a proper distance function $D$ to produce a good proximity matrix. The following section discusses how to design evaluation metrics for the goodness of the computed failure proximity.

2.3 Evaluation Metrics: Requirements and Expectations

An evaluation metric is an indispensable component for any scientific study and so is it for failure proximity. Not only does a good metric provide fair comparison, but it also quantifies the significance of improvement of future studies. In this section, we highlight the requirements and expectations for good metrics and we will describe our proposed metrics in Section 4.

A good evaluation metric should have the following characteristics:

1. It should be objective because human judgment is usually subjective.
2. It should not depend on how the failure proximity is calculated. Instead, it should only be based on the resulting proximity. For example, the evaluation should not depend on what failure signatures are extracted from failures and what distance functions are used.
3. It should not rely on any external applications of the resulting proximity because failure proximity is useful for many tasks such as failure classification, failure clustering, duplicate failure removal, and parallel debugging. Each task has its own quality measure and it is hard to decide which one should be the “ultimate” application.

Therefore, we propose evaluating the goodness of a computed failure proximity purely based on the proximity matrix $M$. The metric should especially account for the following two aspects:

- **cohesion**: to what extent failures in the same class are close to each other, and
- **separation**: to what extent failures in different classes are separated from each other.

We will discuss our proposed metric with details in Section 4.

3 Six Representative Failure Proximities

This section discusses six representative failure proximities, which are listed in Table 2. FP-PROXIMITY and ST-PROXIMITY are two approaches known to be good for crashing failures; CC-PROXIMITY and PE-PROXIMITY are two existing approaches investigated in [10]. Finally, DS-PROXIMITY and SD-PROXIMITY are new approaches we propose in this study. Despite their various forms, all of them are instantiations of a fingerprinting function and a distance function, as shown in Table 2. In the following, we explain them one by one and use the code in Fig. 1 as the running example.

---

### Table 2: Six Representative Failure Proximities

<table>
<thead>
<tr>
<th>F: Fingerprinting function</th>
<th>D: Distance function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP-Proximity</td>
<td>Get failure point</td>
</tr>
<tr>
<td>ST-Proximity</td>
<td>Get stack trace</td>
</tr>
<tr>
<td>CC-Proximity</td>
<td>Get code coverage</td>
</tr>
<tr>
<td>PE-Proximity</td>
<td>Get predicate evaluation</td>
</tr>
<tr>
<td>DS-Proximity</td>
<td>Get dynamic slice</td>
</tr>
<tr>
<td>SD-Proximity</td>
<td>Get likely fault location</td>
</tr>
</tbody>
</table>

---

**Execution 1 with z=1:**

```c
2. x = ...; //FAULT 1
3. y = ...; //FAULT 2
4. z = fgetc(...);
5. if(z>0)
6. B (x);
7. else
8. C (y);
9. }
10. writebuf (x);
11. }
12. }
13. void C (int y)
14. }
15. writebuf (y);
16. }
17. }
18. void writebuf (int v)
19. }
20. }
21. flush (...,&v,...);
22. }
```

**Execution 2 with z=0:**

```c
2. x = ...; //FAULT 1
3. y = ...; //FAULT 2
4. z = fgetc(...);
5. if(z>0)
6. B (x);
7. writebuf (x);
8. C (y);
9. writebuf (y);
10. flush (...,&v,...);
```

**Stack Trace**

- [A]
- [A]
- [A B writebuf]
- [A B]
- [A C]
- [A C writebuf]
Example 1. The program in Fig. 1 consists of four functions. Function A calls either function B or function C, depending on the input value z. Functions B and C use a common output function write2buf to put the given value into a buffer and the buffer is flushed once it is full (line 21). Suppose there are two bugs in lines 2 and 3, respectively, and the value of z determines which fault manifests at the output point (line 21). Two executions with z = 1 and z = 0 and the corresponding stack traces (STs) at each step are listed to the right.

3.1 FP-Proximity: Failure Point-Based Failure Proximity

Failure points are probably the most intuitive choice of failure signature because they are the crashing venue for crashing failures. Before more involved discussion, let us first define failure points and related terminologies.

Definition 1 (Execution Instance). A program P contains multiple program statements and each statement s can be executed multiple times within one execution of the program P. The ith execution of the statement s is called the ith execution instance of the program statement s and is denoted by si.

Definition 2 (Failure Point (FP)). The failure point of a failing execution is the execution instance s0, if s0 is the first execution instance that differs from the expectation.

The above definition depends on what is observable and what is the expectation. For a crashing failure, the failure point is the crashing venue because the crash is the first observable and unexpected behavior. For a noncrashing failure, the failure point is the output point at which the first unexpected output is emitted. The unexpected output is caught by a program oracle, which specifies the expected behavior. This study assumes the existence of such an oracle because failures cannot be otherwise identified. Take Fig. 1 for an example; the execution instance 21 is the failure point for both failures because it emits the first unexpected output.

In FP-PROXIMITY, the fingerprinting function F is to extract the failure point from every failure and the distance function is the 0-1 distance, which is defined as

\[ D(u, u') = \begin{cases} 
0 & \text{if } u \text{ and } u' \text{ correspond to the same statement}, \\
1 & \text{otherwise}, 
\end{cases} \]

where u and u' are two execution instances, representing the failure points of two failures in the context of FP-PROXIMITY.

While the same failure point usually implies the same bug for crashing failures, it is generally not the case for noncrashing failures. Instead, different noncrashing failures can easily share the same failure point because there are usually a limited number of output points even in a complex program. The output content can be erroneous in different ways (because of different faults), but the output point (and, hence, the failure point) is likely the same. Therefore, the failure point is likely no longer discriminative for noncrashing failures and we will provide supportive evidence in Section 5.3.1.

3.2 ST-Proximity: Stack Trace-Based Failure Proximity

As checking what functions are in the call stack when a failure happens is an effective debugging tactic, another speculation is to calculate the failure proximity based on failure stack traces. Let a function call site be the place at which a function is called. A stack trace is formally defined as follows:

Definition 3 (Stack Trace (ST)). The stack trace with respect to an execution instance si, denoted by ST(si), is an ordered list of function call sites, [e1, e2, ..., er], where si is executed in the function called at site er and the call site ej+1 is in the function called at ej for j = 1, 2, ..., r-1.

We name the stack trace-based proximity ST-PROXIMITY and the failure stack trace of a failure is calculated with respect to the failure point of the failure. As shown in Fig. 1, the stack trace is [A B write2buf] and [A C write2buf] for the two failures, respectively. Therefore, the fingerprinting function F of ST-PROXIMITY is to extract the failure stack trace from the failing execution trace.

The extraction of stack trace can be achieved through an offline backward traversal over the control flow trace, which captures the flow of instructions that have been executed. But, in practice, the extraction can be accomplished in an online fashion: detecting unexpected output on the fly and recording the stack trace when the first wrong output is found. Since the goal of this paper is not about optimized implementations, we used a suboptimal offline traversal algorithm to collect stack traces in experiments.

ST-PROXIMITY also uses the 0-1 distance to calculate the proximity between failures. Specifically, it assigns a zero distance to failures with the same failure stack trace and one otherwise. Alternatively, one may treat a stack trace as a sequence of call sites and use some finer level distances (e.g., the edit distance [19]) to quantify the similarities between stack traces. We refrain from doing so because the insertion, deletion, and replacement operations defined in the edit distance barely make sense for stack traces. But, we do not exclude the possibility that the edit distance or other distances may render better results.

3.3 CC-Proximity: Code Coverage-Based Failure Proximity

The third technique we study is CC-PROXIMITY, which populates the distances between failures on the code coverage of executions. This shares the same principle as [10], in which the coverage is computed at the function level. In this study, we define code coverage (CC) as follows:

Definition 4 (Code Coverage (CC)). The code coverage of an execution e of a program P is the set of program statements that are ever executed in e.

In Fig. 1, the code coverage is {2, 3, 4, 5, 6, 11, 21} for Execution 1 and {2, 3, 4, 5, 8, 16, 21} for Execution 2. Note that code coverage is also called an execution slice [20].

Because code coverage is essentially a set of executed statements, any distances defined on sets suffice. We choose here the Jaccard distance, which is proposed by Levenshowsky and Winter [21].
The Jaccard distance is a metric [21], i.e., satisfying the four properties discussed in Section 2.2. In summary, the fingerprinting function of CC-PROXIMITY is to track the code coverage and the distance function is the Jaccard distance. Note that multiple instances of the same statement are represented by a single statement in the same code coverage.

### 3.4 PE-Proximity: Predicate Evaluation-Based Failure Proximity

PE-PROXIMITY is based on predicate evaluation, which is another way of characterizing executions [22], [23], [24]. In general, a predicate is a proposition about any program properties and many kinds of predicates can be instrumented, depending on the need and overhead expectation. In this study, the following two kinds of predicates are uniformly instrumented in subject programs because they were shown effective in characterizing executions by previous studies (e.g., see [23], [24]):

- **[boolean].** For each Boolean expression \( b \), a predicate \( b == \text{true} \) is instrumented.
- **[return].** For each function call site, three predicates, \( "r > 0," \) \( "r = 0," \) and \( "r < 0" \) are instrumented, where \( r \) is the function return value.

The source code location at which a predicate \( P \) is instrumented is called the instrumentation site of the predicate. Every time an instrumentation site is executed, the corresponding predicate evaluates as either true or false. In PE-PROXIMITY, the fingerprinting function \( F \) transforms the collected predicate evaluations into a predicate vector, which is defined as follows:

**Definition 6 (Predicate Evaluation Vector).** Suppose \( L \) predicates are instrumented in a program \( P \), numbered in any arbitrary order fixed across all runs. The predicate evaluation vector of an execution is an \( L \)-dimensional vector \( v \), where the \( i \)th dimension \( v(i) \) is the ratio of \text{true} evaluations of the \( i \)th predicate \( P_i \) over the total number of evaluations during the execution. If \( P_i \) is not ever evaluated during the execution, \( v(i) = 0.5 \) as no evidence indicates whether the evaluation of \( P_i \) is biased to \text{true} or \text{false}.

Since predicate vectors are numeric vectors, PE-PROXIMITY uses the euclidean distance as the distance function \( D \), which is a special case \((p = 2)\) of the \( p \)-norm Minkowski distance [25]:

\[
D_p(v, v') = \left( \sum_{i=1}^{L} |v(i) - v'(i)|^p \right)^{1/p},
\]

where \( v \) and \( v' \) are instantiated with the predicate evaluation vectors of two failures in the context of PE-PROXIMITY.

### 3.5 DS-Proximity: Dynamic Slicing-Based Failure Proximity

Dynamic slicing, first invented as a debugging aid [26], is able to identify the subset of program statements that are involved in producing a program failure. Dynamic slicing techniques observe a program execution, collect the dependences between executed statements, and, finally, compute dynamic slices from the collected dependences. Specifically, a dynamic slice includes both dynamic data dependences and dynamic control dependences, which are formally defined as follows:

**Definition 7 (Dynamic Data Dependence).** An execution instance \( s_i \) of the statement \( s \) has a data dependence \((dd)\) on the execution instance \( t_j \) of the statement \( t \), denoted by \( s_i \xrightarrow{dd} t_j \), if and only if there exists a variable \( var \) whose value is defined at \( t_j \) and then used at \( s_i \).

**Definition 8 (Dynamic Control Dependence).** A statement \( s_i \) of statement \( s \) has a control dependence \((cd)\) on the execution instance \( t_j \) of statement \( t \), denoted by \( s_i \xrightarrow{cd} t_j \), if and only if

1. statement \( t \) is a predicate statement, and
2. the execution of \( s_i \) is the result of the branch outcome of \( t_j \).

**Definition 9 (Dynamic Slice).** The dynamic slice of the \( i \)th execution instance of statement \( s \), denoted by \( DS(s_i) \), is

\[
DS(s_i) = \{s\} \cup \bigcup_{\forall t_j, s_i \xrightarrow{dd} t_j \text{ or } s_i \xrightarrow{cd} t_j} DS(t_j).
\]

The fingerprinting function \( F \) of DS-PROXIMITY is to compute the dynamic slice from the FP; thus, the dynamic slice contains all statements that directly or indirectly contribute to the program failure which are either erroneous outputs or program crashes. For example, the dynamic slice for the execution with \( z = 1 \) in Fig. 1 is \( \{2, 4, 5, 6, 11, 21\} \). Please note that, even though dependences are defined between statement instances, only unique statements are included in the slice. Now that the failure signature of DS-PROXIMITY is essentially a set of statements, the same Jaccard distance as defined in Definition 5 can also serve as the distance function \( D \) for DS-PROXIMITY.

### 3.6 SD-Proximity: Statistical Debugging-Based Failure Proximity

This section presents SD-PROXIMITY, which utilizes statistical debugging algorithms to extract failure signatures. We discuss its fingerprinting function and its distance function in Sections 3.6.1 and 3.6.2, respectively, and describe how SD-PROXIMITY could help failure assignment in Section 3.6.3.

#### 3.6.1 Fault-Aware Fingerprinting

The root cause is apparently the optimal failure signature, but it is too expensive to obtain it for all failures. We therefore ask whether root causes can be automatically approximated. Encouraged by recent advances in fault localization (a.k.a. automated debugging) studies [23], [24], [27], [28], [29], [30], [31], we believe in an affirmative answer.
Theoretically, any fault localization algorithm can be used to find the likely fault location. We use SOBER [24] in this study because of its immediate availability to us. Readers are encouraged to explore the feasibility with other fault localization algorithms.

The debugging algorithm SOBER localizes software faults by contrasting the set of failures X against a set of passing traces \( Y = \{ y_1, y_2, \ldots, y_n \} \). Specifically, the subject program is first instrumented with the “boolean” and “return” predicates, as described in Section 3.4. Then, SOBER checks how differently each predicate evaluates in failing executions X than it does in passing executions Y. The larger the difference, the higher the predicate is ranked in the output ranking \( \tau \) of all instrumented predicates, i.e., \( \tau = \text{SOBER}(X, Y) \). Let \( \tau(P_i) \) denote the position of predicate \( P_i \) in \( \tau \); then, we say predicate \( P_i \) ranks higher than predicate \( P_j \) if \( \tau(P_i) < \tau(P_j) \) and lower otherwise. Instrumentation sites of higher ranked predicates are regarded as the likely fault location by SOBER. Readers interested in the details about SOBER are referred to [24].

As one may have noticed, SOBER is not restricted to contrasting \( X \) against \( Y \) in their entireties. Instead, any subsets of \( X \) and \( Y \) can be contrasted for fault localization. As an extreme case, each failure \( x_i \in X \) can be contrasted against \( Y \), generating a predicate ranking \( \tau_i \), i.e.,

\[
\tau_i = \text{SOBER}(\{x_i\}, Y) \quad (i = 1, 2, \ldots, n),
\]

and \( \tau_i \) is called the individual ranking for failure \( x_i \).

Each individual ranking \( \tau_i \) may not always pinpoint the root cause of the failure \( x_i \) (depending on the quality of SOBER, \( x_i \), and \( Y \)), but it does suggest what predicates \( x_i \) suggests as more fault relevant than others. We call the transformation described by (1) fault-aware fingerprinting because \( \tau_i \) embodies failure \( x_i \)'s opinion on the likely fault location. Therefore, SD-PROXIMITY uses the statistical debugging algorithm SOBER as the fingerprinting function and the extract failure signature for each failure is the individual ranking. Then, what is needed is a distance function that quantifies the agreement between two predicate rankings. We propose a weighted Kendall’s tau distance for this purpose, as is discussed in the following section.

### 3.6.2 Weighted Kendall’s Tau Distance

As its name suggests, the weighted Kendall’s tau distance is a variant of the conventional Kendall’s tau distance [32]. We propose it to accommodate the fact that not all predicates are equally important. Letting \( \pi \) and \( \sigma \) be two rankings of \( L \) predicates, the conventional Kendall’s tau distance \( D_K(\pi, \sigma) \) is defined as

\[
D_K(\pi, \sigma) = \sum_{1 \leq i < j \leq L} K(P_i, P_j),
\]

where

\[
K(P_i, P_j) = \begin{cases} 1 & \text{if } [\pi(P_i) - \pi(P_j)] \ast [\sigma(P_i) - \sigma(P_j)] < 0, \\ 0 & \text{otherwise}, \end{cases}
\]

and “\( \ast \)” is the regular multiplication. We say predicates \( P_i \) and \( P_j \) constitute a discordant pair if their relative orders in \( \pi \) and \( \sigma \) disagree. Kendall’s tau distance essentially counts the number of discordant pairs between \( \pi \) and \( \sigma \). For example, suppose \( \pi = (P_1, P_2, P_3) \) and \( \sigma = (P_2, P_3, P_1) \); then, \( D_K(\pi, \sigma) = 2 \).

Although Kendall’s tau distance is a reasonable distance for general rankings, it needs adjustment to quantify the disagreement between predicate rankings. First, because predicates are uniformly instrumented, many of them are irrelevant to the fault. Discordant pairs formed by these irrelevant predicates should be excluded in distance computation. Second, even after fault-irrelevant predicates are excluded, not all remaining predicates are equally important. We therefore propose a predicate weighting schema to address these two concerns.

The weighting schema is comprised of two heuristics. The first one is to derive predicate weights from the ranking \( \tau = \text{SOBER}(X, Y) \). Explicitly, if the top-\( k_1 \) predicates of \( \tau \) are taken as fault relevant, the weight of predicate \( P_i \) is then

\[
W^k_1(P_i) = \frac{I(k_1 - \tau(P_i))}{k_1},
\]

where \( I(t) \) is an indicator function that equals to one if \( t \geq 0 \) and zero otherwise. If a predicate ranks lower than \( k_1 \) in \( \tau \), it gets a weight of zero at this step. Here, we assign an equal weight to the top-\( k_1 \) predicates, although decaying weights are also possible.

The second weighting heuristic is to derive the predicate weights from individual rankings \( \tau_i \)'s. Specifically, we take predicates that are within the top part of more individual rankings as more fault relevant because, intuitively, the \( n \) individual rankings \( \tau_i \)'s are like \( n \) votes for fault-relevant predicates. Explicitly, if top-\( k_2 \) predicates are considered, the weight of predicate \( P_i \) is

\[
W^k_2(P_i) = \sum_{t=1}^{m} \frac{I(k_2 - \tau_i(P_i))}{m k_2},
\]

where \( I(t) \) is the same indicator function. Combining these two heuristics, the final weight of predicate \( P_i \) is

\[
W(P_i) = (1 - \alpha)W^k_1(P_i) + \alpha W^k_2(P_i),
\]

where \( \alpha \) is the parameter balancing the two components. The parameters \( k_1 \) and \( k_2 \) are set based on one’s confidence in how many top predicates are fault relevant in the global ranking and individual rankings, respectively. By default, the parameters \( k_1, k_2 \), and \( \alpha \) are 10, 1, and 0.1, respectively, and they are used throughout the experiment in this study. Readers interested in the effects of different parameter settings and the necessity of predicate weighting are referred to [33].

After deriving the weight of each predicate, we propose the following weighted Kendall’s tau distance to calculate the pairwise distance in SD-PROXIMITY:

**Definition 10 (Weighted Kendall’s Tau Distance).** Given \( \pi \) and \( \sigma \), two rankings of the \( L \) predicates, the weighted Kendall’s tau distance \( D_{W,K} \) is

\[
D_{W,K}(\pi, \sigma) = \sum_{1 \leq i < j \leq L} K(P_i, P_j) W(P_i, P_j),
\]
where \( K(P_i, P_j) \) is the same as that in (2) and \( W(P_i, P_j) = W(P_i)W(P_j) \).

We proved in [33] that the weighted Kendall’s tau distance is a distance metric.

### 3.6.3 Support for Failure Assignment

A distinct advantage of SD-PROXIMITY is its natural support for failure assignment. When the distances between a set of failures are small as calculated by SD-PROXIMITY, it is certain that these failures agree on the likely fault location. When the consensus about the likely fault location, or the consensus location in short, is found, these failures can be assigned to the developer in charge of the consensus location. The critical question then is how to find the consensus location.

We use the following algorithm to find the consensus location for SD-PROXIMITY. Given a set of failures with mutually small distances calculated by SD-PROXIMITY, we count, for each predicate, the number of individual rankings in which this predicate is within the top-\( k \). Let us call this number the top frequency of the predicate; then, we can rank all predicates in descending order of their top-\( k \) frequencies. The source code pointed by the top-\( k \) predicates in the final ranking is regarded as the consensus location. The parameter \( k \) reflects one’s belief in how many top predicates are fault relevant. According to previous studies [24], [34], \( k = 5 \) is a good choice and it is the value used throughout this study.

### 3.7 Other Failure Proximities

Although we have discussed six proximities in the above, we want to emphasize here that there are many other instantiations of the fingerprinting function and the distance function and each instantiation will render a failure proximity.

Choosing the proper fingerprinting function \( F \) is the critical component of failure indexing. Once the failure signature is determined, the distance function \( D \) is mostly determined because the signature usually falls into some of the following categories: single items, sets, numeric vectors, and rankings. For each category, some standard distance functions will apply. For example, the 0-1 function is the standard choice for single items, the Jaccard distance for sets, the euclidean distance for numeric vectors, and Kendall’s tau distance for rankings. But, one needs to note that many variants may exist for a given distance, e.g., the Minkowski distance varies with different values of \( p \).

Furthermore, we want to emphasize the numerous possibilities of fingerprinting failures through debugging algorithms. In general, any debugging algorithms can be used for failure fingerprinting and this paper only shows the possibility of using SOBER for this purpose. The plethora of debugging algorithms [23], [24], [27], [28], [29], [31] suggests that many possibilities can be explored with potentially better results.

In order to use a debugging algorithm for failure proximity, what is needed is 1) a distance function that quantifies the difference between two debugging results and 2) a mechanism to find the consensus location for failure assignment. As a concrete case, let us consider how to leverage Delta Debugging (DD) [28], a well-known debugging algorithm, to populate a failure proximity. First, using DD as the fingerprinting function, the failure signature is extracted as a set of source code locations. Then, the Jaccard distance can be adopted to calculate the proximity matrix and this completes a new failure proximity, which can be similarly named DD-PROXIMITY: Delta Debugging-based failure proximity. In order to support failure assignment, the consensus location can be found through set intersections, for instance.

In summary, failure proximity is a research topic in which many possibilities are yet to be explored. It is unrealistic (and unnecessary) to enumerate all of the possibilities in a single study. In the following section, let us study how to evaluate and compare different proximities.

### 4 Evaluation Metrics

This section presents three evaluation metrics that are designed for different evaluation needs: 1) The Silhouette Coefficient scores the goodness of different proximities quantitatively (Section 4.1), 2) the proximity graph facilitates qualitative comparison of different proximity results (Section 4.2), and 3) the A-score is the first measure of the accuracy of failure assignment (Section 4.3). All three of the metrics can be automatically computed.

#### 4.1 Silhouette Coefficient: A Quantitative Metric of the Goodness of Proximity

The Silhouette Coefficient (SC) was originally designed to evaluate the internal structure of data clustering results without knowing what data should be clustered together [15]. In the setting of failure proximity, because what failures should be close to each other and what should not are already known for evaluation purposes, we choose to amend the Silhouette Coefficient to quantify the goodness of a proximity result. Intuitively, the optimal proximity assigns a zero distance between failures due to the same fault and a nonzero value otherwise. The amended Silhouette Coefficient quantifies how far a proximity result deviates from the optimal proximity by considering both cohesion and separation simultaneously.

The amended Silhouette Coefficient for each failure \( x_i \) is defined by

\[
SC(x_i) = \frac{b_i - a_i}{\max(a_i, b_i)},
\]

where

\[
a_i = \frac{\sum_{j \in C_k(x_i)} M(i, j)}{|C_k(x_i)|}
\]

and

\[
b_i = \min_{k=1,2,...,m,k \neq \Phi(x_i)} \frac{\sum_{j \in C_k} M(i, j)}{|C_k|}.
\]

Recall that \( M \) is the proximity matrix, \( \Phi(x_i) \) returns the index of the fault that accounts for the failure \( x_i \), and \( C_k \) denotes the set of all failures due to the \( k \)th fault. In the above formulas, \( a_i \) is the average distance from \( x_i \) to all
other failures in the same failure class. To compute $b_{i}$, we first calculate the average distances between $x_{i}$ and failures in $C_{k}$ for all $k \neq F(x_{i})$ and $b_{i}$ is the minimum of the above $m - 1$ average distances.

$SC(x_{i})$ ranges inclusively from $-1$ to $+1$. A positive value means $x_{i}$ is closer to failures in the same class than those in different classes and vice versa for a negative value. Given a proximity matrix $M$, the overall Silhouette Coefficient is

$$SC(M) = \frac{\sum_{i=1}^{n} SC(x_{i})}{n}.$$  

Again, $SC(M)$ ranges from $-1$ to $1$ and a higher value indicates a better proximity result. Finally, let us use the following example to further illustrate the implication of Silhouette Coefficient.

**Example 2.** Suppose there are six failures $\{x_{1}, x_{2}, \ldots, x_{6}\}$ due to two faults, namely, $\Phi(x_{1}) = \Phi(x_{2}) = \Phi(x_{4}) = 1$, and $\Phi(x_{3}) = \Phi(x_{5}) = \Phi(x_{6}) = 2$. In Fig. 4a, we use crosses and circles to represent failures in failure classes $C_{1}$ and $C_{2}$, respectively. Fig. 2a visualizes the optimal proximity result (more discussion about visualization in Section 4.2) and Fig. 2b plots a suboptimal proximity case, where the failure $x_{2}$ deviates from its class $C_{1}$.

According to (7) and (8), for the suboptimal case (Fig. 2b), $a_{2} = 4$, $b_{1} = 1$, and $SC(x_{2}) = -0.75$, which reflects $x_{2}$'s deviation from its failure class. Similarly, we can calculate $SC(x_{1}) = SC(x_{4}) = 0.6$ and $SC(x_{3}) = SC(x_{5}) = SC(x_{6}) = 1$, so the overall Silhouette Coefficient of the suboptimal proximity is 0.575.

### 4.2 Proximity Graph: A Qualitative Metric of the Goodness of Proximity

Although the Silhouette Coefficient provides a quantitative way of assessing the goodness of a proximity result, usually, a human analyst wants to have an intuitive picture of the proximity of a (potentially huge) set of failures. In this section, we explain a visual presentation of failure proximity, which complements the above Silhouette Coefficient.

Following the requirement of the evaluation metric as laid out in Section 2.3, the visualization should only depend on the proximity matrix $M$. We therefore adopt the multidimensional scaling (MDS) techniques [35], which are also used in previous work [36].

Given $n$ failures whose pairwise distances are calculated in whichever way as described in Section 3, their proximity is encoded entirely in the proximity matrix $M$. As the distances are usually computed in a high-dimensional space, the proximity cannot be visualized in the original space. Instead, the MDS techniques arrange the $n$ failures (represented as points) in a much lower (usually two or three) dimensional space to best preserve the pairwise distances encoded in $M$. Readers interested in the technical details of MDS are referred to [35].

The result from MDS is a proximity graph, which visualizes the proximity between failures. Since the objective of MDS techniques is to best preserve the original distances in the low-dimensional space, the orientations of axes are arbitrary, i.e., the orientation of axes does not change the pairwise distances. But, it is worth mentioning that the scale of the axes is relevant because it shows the actual distance a unit length represents in the graph. Therefore, a caveat about interpreting proximity graphs is that a proximity graph is not a projection of the original data into a low-dimensional subspace and no projection should be applied to proximity graphs, for example, to squeeze distant points together.

### 4.3 A-Score: A Quantitative Metric of Failure Assignment Accuracy

Without loss of generality, given a cluster of $n'$ failures $X' = \{x_{1}, x_{2}, \ldots, x_{n'}\}$ that are due to $m'$ faults $F' = \{f_{1}, f_{2}, \ldots, f_{m'}\}$, the fault $f_{p}$ is the primary fault for the failure cluster $X'$ if it causes most of the failures in $X'$, namely,

$$f_{p} = \arg\max_{f \in F'} \{|x| \ x \in X, \text{ and } \Phi(x) = f\},$$  

where $\arg\max_{z \in Z} T(z)$ returns the element in the set $Z$ that maximizes the function $T(z)$.

Because the consensus location is usually a set of program statements, we propose the following $A$-score to quantify the assignment accuracy:

1. Compute the program dependence graph (PDG) $G$ for the subject program, where vertices and undirected edges between vertices represent program statements and static data/control dependences in between, respectively.
2. Vertices corresponding to the primary fault $f_{p}$ are marked as $defect$ and the set of defect vertices is written as $V_{defect}$.
3. Vertices corresponding to the consensus location are marked as $blamed$ and the set of blamed vertices is written as $V_{blamed}$.
4. Perform a breadth-first search on $G$ from $V_{blamed}$ until any vertices in $V_{defect}$ are touched. The set of vertices covered by the breadth-first search is denoted by $V_{examined}$.
5. The $A$-score, defined below, estimates the percentage of code that needs to be examined before locating the primary fault:

$$A = \frac{|V_{examined}|}{|G|} \times 100\%,$$  

where $|G|$ denotes the size of the PDG $G$.

Readers familiar with automated debugging research will immediately identify the same principle shared by the above $A$-Score and the $T$-score that is widely used in evaluating fault localization techniques [23], [24], [27], [28],
Similarly to the $T$-score, the $A$-score objectively quantifies the closeness between the consensus location and the primary fault location. Explicitly, a smaller $A$-score means that the consensus location is closer to the primary fault location and, consequently, the primary fault is easier to fix if the developer in charge of the consensus location is assigned. We will use the $A$-score to evaluate the accuracy of failure assignment for SD-PROXIMITY in Section 5.3.

5 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of the six proximities discussed in Section 3 using the above proposed metrics. We first describe the experimental setup in Section 5.1 and then give an overview of the experimental result in Section 5.2. Section 5.3 investigates a set of five research questions by examining the experimental results in detail. Finally, Section 5.4 summarizes the lessons we have learned from the experiments.

5.1 Experiment Setup

We obtained three subject programs, flex, grep, and gzip, together with their accompanying test suites from the “Software-artifact Infrastructure Repository” (SIR) [37], which is “a repository of software-related artifact meant to support rigorous controlled experimentation.” The version number, physical Source Lines of Code (SLOC), and the size of the accompanying test suite are listed in Table 3. The last four columns in Table 3 will be explained later. The two faults in flex are injected by the SIR researchers and the other two faults for each grep and gzip, planted by the authors, are the same as used in previous studies [24], [31], [33], [34]. The six faults are selected because 1) the test suites we have regarding the subject programs are adequate to provide enough failure cases for each fault and 2) failures caused by the two faults in the same program do not significantly overlap so that the ambiguity can be minimized in evaluation. Note that this paper does not discuss failure proximity for faults that are semantically correlated.

Table 3. Characteristics of the Three Subject Programs

| Program | SLOC | Test # | Fault 1 | Fault 2 | Failure # | $|C_1| | $|C_2| | $|C_1 \cap C_2| |
|---------|------|--------|---------|---------|------------|-----|-----|-----|-----|
| grep-2.2 | 15,633 | 470 | Off-by-one | Subclause-missing | 136 | 48 | 88(81) | 0 |
| gzip-1.2.3 | 6,184 | 217 | Subclause-missing | Subclause-missing | 82 | 65 (54) | 17 (16) | 0 |
| flex-2.4.7 | 9,212 | 525 | Off-by-one | Off-by-one | 255 | 163 | 92 | 0 |

Details about the six faults in grep, gzip, and flex are presented in Figs. 3, 4, and 5, respectively. All six faults are semantic bugs that do not incur crashes. Although one may speculate that failures due to the same fault can be identified from outputs, we can personally certify the difficulty even with the three mid-sized programs.

The experimental evaluation requires knowing the ground truth of what failures are due to the same fault. Ideally, experienced developers can manually examine every failure and determine the culprit fault (or faults), but this is apparently too expensive. We therefore propose the following rule to determine the culprit.

- If a test case fails at the existence of multiple faults, the culprit for the failure is the faults that can each cause the test case to fail.

For example, if there are two faults in a program, the culpability can be determined by Table 4.
Table 4 lists the eight possible situations when different faults are activated in combination and the last column lists the culprit. Because culprits are only convicted for failures, there are no culprits for Situations 1 through 4. The culprit for Situation 5 is the coexistence of both faults because neither fault can cause the execution to fail by itself, but they can do so only through working together. For Situations 6 and 7, Fault 1 and Fault 2 are convicted as the culprit, respectively, according to the aforementioned rule. Finally, both faults are the culprit for Situation 8 because the test case fails whenever either or both faults are activated.

Although there are eight possible situations in total, we only observe Situations 1, 6, and 7 in the experiment. Situations 2 to 5 are not observed because they are intrinsically small-probability events. For example, in Situations 2 to 4, failures disappear once both faults are activated. While Situation 8 is more likely to be observed than Situations 2 to 5, no failures fall into Situation 8 in our experiment. Therefore, one and only one fault is convicted for each failure in this study. The sets of failures due to

<table>
<thead>
<tr>
<th>Situation</th>
<th>Fault 1 On</th>
<th>Fault 2 On</th>
<th>Both On</th>
<th>Culprit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pass</td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Fail</td>
<td>Pass</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>Fail</td>
<td>Pass</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>Fail</td>
<td>Fail</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>5</td>
<td>Pass</td>
<td>Fail</td>
<td></td>
<td>Coexistence</td>
</tr>
<tr>
<td>6</td>
<td>Fail</td>
<td>Pass</td>
<td></td>
<td>Fault 1 Only</td>
</tr>
<tr>
<td>7</td>
<td>Pass</td>
<td>Fail</td>
<td></td>
<td>Fault 2 Only</td>
</tr>
<tr>
<td>8</td>
<td>Fail</td>
<td>Fail</td>
<td></td>
<td>Both Faults</td>
</tr>
</tbody>
</table>

Fault 1: An off-by-one error in deflate.c

Fault 2: Another subclause missing error in deflate.c

Fault 1: An off-by-one error in main.c

Fault 2: An off-by-one error in gen.c
Fault 1 and Fault 2 are denoted by $C_1$ and $C_2$, respectively, and their sizes and the intersection size are listed in Table 3.

With little difficulty, Table 4 can be generalized to cases with more than two faults, i.e., $2^{m+1}$ situations for a program with $m$ faults. But, one needs to note that such culprit determination is only needed for in-house evaluation but not for practical usage.

Finally, we briefly describe some details about our implementation. Fig. 6 depicts the instrumentation framework that is used by FP-PROXIMITY, ST-PROXIMITY, CC-PROXIMITY, and DS-PROXIMITY. Specifically, we use this framework to extract FPs, STs, CC, and dynamic slices. The framework consists of a static component and a dynamic component. The static component computes control flowgraphs and static control dependence that are required to instrument a program binary. The static analysis is implemented on top of Diablo [38], a retargetable link-time binary rewriting framework. Diablo has the capability of constructing the control flowgraphs for a binary. Static control dependence is computed from the control flowgraph. This static information is indexed by virtual addresses such that it can be shared by both the static and dynamic components. The dynamic component of the system, which is based upon the Valgrind [8] tool, accepts the same binary and dynamically instruments it by calling the WET instrumenter [39]. The instrumented code is executed with the support of the WET runtime, which consists of a set of callback functions handling certain events (e.g., entering functions, accessing memory, arithmetic operations, and predicates). The WET instrumenter and runtime are developed to enable a collection of dynamic dependence information together with dynamic control flow information, which are later used in computation of failure points, stack traces, code coverage, and dynamic slices. Note that if a program fails without emitting any outputs, the corresponding failure points, dynamic slices, etc., cannot be computed. In Table 3, the numbers in parentheses are the counts of failures with nonempty outputs. Although this framework can also collect predicate evaluations, PE-PROXIMITY and SD-PROXIMITY actually use the same instrumenter as in our previous studies [24], [31], [34] for quick prototyping. Once failure signatures are derived, all the rest, which include calculating the distances, computing the SC, and drawing the proximity graph, are automatically processed in Matlab.

5.2 Result Overview

This section reports on the experimental result with CC-PROXIMITY, PE-PROXIMITY, DS-PROXIMITY, and SD-PROXIMITY. We postpone the detailed discussion of FP-PROXIMITY and ST-PROXIMITY to Sections 5.3.1 and 5.3.2, respectively.

Figs. 7, 8, and 9 present the proximity comparison on grep, gzip, and flex, respectively. In each figure, the proximity graphs and corresponding SCs (the numbers inside parentheses) for different proximity techniques are listed for comparison.

In each proximity graph, failures in $C_1$ and $C_2$ are denoted by crosses and circles, respectively. Ideally, crosses and circles should be separated from each other but themselves be densely clustered, giving a Silhouette Coefficient of one.

Now, let us take a close look at the result with grep (Fig. 7). First, the result with CC-PROXIMITY exhibits both low separation and low cohesion (Fig. 7a), where failures
due to Fault 2 (circles) are separated into two loose clusters with crosses scattered and overlapping with circles. Fig. 7b shows that the result becomes better with PE-PROXIMITY, but the cohesion is still low, e.g., the circles stretch in a long line, which indicates that failures from the same fault can have quite divergent profiles. DS-PROXIMITY improves over CC-PROXIMITY by removing profiles irrelevant to program failures and hence obtains a much cleaner proximity graph and a considerably higher Silhouette Coefficient (Fig. 7c). Finally, SD-PROXIMITY renders the best result among the four: All circles collapse together and crosses form several dense clusters. This improvement can be attributed to the mechanism of SD-PROXIMITY: Although failures in $C_2$ (circles) exhibit divergent profiles, their suggested fault locations, as discovered by SOBER, are roughly the same. This good property is also observed in the experimental results with gzip and flex (Figs. 8 and 9).

While SD-PROXIMITY achieves the best SC on all of the three case studies, the results of CC-PROXIMITY, PE-PROXIMITY, and DS-PROXIMITY are not necessarily bad in all aspects, as indicated by the proximity graphs in Figs. 7, 8, and 9. For example, the result of CC-PROXIMITY, as shown in Fig. 8a, is very attractive and may even appear better than that of SD-PROXIMITY (Fig. 8d). But then, why is CC-PROXIMITY’s Silhouette Coefficient not as good as that of DS-PROXIMITY? The explanation lies in the fact that failures due to gzip’s first fault are split by CC-PROXIMITY into two subsets of comparable sizes and this split is penalized by the Silhouette Coefficient because such a split jeopardizes both cohesion and separation. Recall that the best result credited by the Silhouette Coefficient is that failures due to the same fault are indefinitely close to each other while being far from failures due to other faults (as illustrated in Fig. 2a).

On the other hand, SD-PROXIMITY’s result is not perfect either, despite its high Silhouette Coefficients. For example, there are some crosses near the cluster in Fig. 7d. Such impurity is expected because it is unrealistic to expect any automated and practical approaches to always obtain the optimal result. So far, the only way to warrant the optimal result is to manually debug every failure, which is unfortunately too manually intensive to be practical.

5.3 Research Questions
This section investigates five research questions that readers may have about the properties of different proximities, for instance, whether good failure signatures for crashing failures are still good for noncrashing failures, what contributes to a good proximity technique, etc.

5.3.1 Research Question I: Is FP-PROXIMITY Effective for Noncrashing Failures?
We applied FP-PROXIMITY to the three subject programs and the result showed that failure points are not good failure signatures for noncrashing failures, as explained below. We found that all of the 70 failures on gzip have the same failure point, namely, line 156 of util.c in function write_buf. It is also the case for flex, where all of the 255 failures share the same failure point at line 711 of misc.c in function skelout. Interestingly, FP-PROXIMITY is pretty good for grep. All of the 81 failures in $C_2$ and four failures in $C_1$ have the same failure point (line 1,033 of grep.c in function main). The remaining 44 failures in $C_1$ have two different failure points. The overall stack trace is 0.5888, which is even slightly better than that of SD-PROXIMITY (0.513). These dichotomous results across different subject programs suggest that FP-PROXIMITY is a brittle technique.

5.3.2 Research Question II: Is ST-PROXIMITY Effective for Noncrashing Failures?
The result with ST-PROXIMITY is very similar to the result with FP-PROXIMITY. The 255 failures on flex incur the same stack trace and so do the 70 failures on gzip.

Fig. 10 lists the stack traces shared by all of the failures for gzip and flex, respectively. For each program, the first column shows the stack depth with corresponding program counters (PCs) in hex format; the next two columns list the function names and their source code locations.
Fig. 11. Fault relevance improves failure proximity.

Consider Fig. 10a, the stack trace for the gzip failures; the function flush_outbuf flushes a global buffer the contains all of the output values. Even though different faults cause different erroneous output values, the stack trace of flushing the output is unfortunately the same.

The case with flex is trickier. The source code line (misc.c:711) is printf("%s
", buf). The symbol "\n" flushes the OS buffer, which is filled up by other printf statements. Again, all failures share the same FP. While this problem can be mitigated by redefining observation points as the statements that fill in the buffer, the programming practice of having one or very few output functions (functions filling the buffer) makes us pessimistic.

As to grep, we found that each of the three unique failure points corresponds to a unique stack trace. Hence, ST-PROXIMITY has the same proximity result as FP-PROXIMITY. The similar results between FP-PROXIMITY and ST-PROXIMITY suggest that failure points and stack traces are brittle failure signatures for noncrashing failures although they are very good for crashing failures.

5.3.3 Research Question III: Does the Fault Relevance of Failure Signatures Improve Proximity?

We conjecture that pruning fault-irrelevant information from execution profiles would improve the resulting proximity. This conjecture is supported by a comparative study of the experiment results, as summarized in Fig. 11.

Fig. 11 collects the Silhouette Coefficients in Figs. 7, 8, and 9 and visualizes the comparison. First, CC-PROXIMITY and PE-PROXIMITY give the worst result among the four because their failure signatures are raw execution profiles and most of the raw profiles are fault irrelevant. It is hard to reason whether CC-PROXIMITY or PE-PROXIMITY is better than the other because code coverage and predicate evaluation are merely different representations of the same execution profile. As shown in Fig. 11, PE-PROXIMITY outperforms CC-PROXIMITY on grep and flex, whereas CC-PROXIMITY wins on gzip.

The DS-PROXIMITY technique significantly improves over CC-PROXIMITY, as evidenced in Fig. 11, because it carefully prunes fault-irrelevant information from raw profiles. For a similar reason, SD-PROXIMITY is much better than PE-PROXIMITY because only fault-relevant predicates are considered for failure proximity calculations.

In Fig. 11, we also notice that SD-PROXIMITY is consistently better than DS-PROXIMITY. A possible reason for this is that, although dynamic slices contain statements contributing to the final failure, most statements, however, are not essential to program failures. For example, many statements in the dynamic slices could also be executed in passing executions. Therefore, a possible improvement over DS-PROXIMITY is to leverage the code coverage in passing executions to prune the dynamic slices computed from failing executions. Since there are many ways to prune dynamic slices [40], [41], we leave them to future studies.

5.3.4 Research Question IV: Is Failure Assignment through SD-Proximity Accurate?

We now examine the accuracy of failure assignment as supported by SD-PROXIMITY. Fig. 12 presents the proximity graphs for SD-PROXIMITY on the three programs. We asked 10 independent human judges to visually identify clusters in each graph and eight clusters were unanimously identified, as shown in Fig. 12. This indicates that the visual identification of clusters is stable and no parameter such as the number of clusters is needed. These nice properties underpin the utility of proximity graphs in supporting failure explorations.

Fig. 12. Identified failure clusters on the three subject programs. (a) Grep-2.2. (b) Gzip-1.2.3. (c) Flex-2.4.7.
Table 5 presents the A-scores for the eight clusters in Fig. 12. For four out of the eight clusters, the A-score is less than 1 percent, which indicates that the consensus location found through SD-PROXIMITY pinpoints the primary fault location. On another three clusters, the A-score is no more than 4 percent, which means the consensus location is in the close vicinity of the primary fault location. As expected, there is no silver bullet: The A-score is 71.3 percent for Cluster 2 of flex.

As one may have noticed, not all failures are assigned to a particular cluster in Fig. 12. These unassigned failures do not matter because 1) they only account for a marginal percentage of all failures and 2) not all failures need to be assigned at once. When some faults are fixed through assigned failures, some unassigned failures may not fail anymore.

### 5.3.5 Research Question V: Is SD-Proximity Computationally Expensive?

Because SD-PROXIMITY uses SOBER to fingerprint every failure, one may be concerned about the fingerprinting cost. In fact, since SOBER only carries numerical computation, it is lightweight even when being invoked many times. For example, in the case study of grep, it took 13.6 seconds to fingerprint the 136 failures on a Pentium 4 machine with 1 Gbyte memory.

### 5.4 Lessons Learned from Experiments

This section highlights three key lessons that we have learned from the above experimental study. We hope this summary will serve as a quick takeaway for the above detailed discussion about experiments. The three lessons are the following:

- **Lesson 1: Noncrashing failures call for different failure signatures than those good for crashing failures.** As we have seen from the above study, neither failure points nor stack traces are effective in distinguishing noncrashing failures due to different faults, although they are good for crashing failures [4], [5], [6]. This suggests that more sophisticated analysis is needed to scrutinize failure executions and to extract signatures that are more related to the root cause. The improved effectiveness of DS-PROXIMITY and SD-PROXIMITY over FP-PROXIMITY and ST-PROXIMITY demonstrates the rewards of such sophisticated analysis and more work is definitely on the way for even better results.

- **Lesson 2: Discriminative failure signatures are the key to good proximities for noncrashing failures.** Although the distance function is also important, what is more critical for a good proximity is how discriminative the chosen failure signature is. As shown in the above study, failure points and stack traces fall short for noncrashing failures because they are not discriminative enough, e.g., failures due to different faults can easily emit erroneous outputs at the same location. CC-PROXIMITY and PE-PROXIMITY improve over FP-PROXIMITY and ST-PROXIMITY because code coverage and predicate evaluations are more discriminative as they include information regarding how erroneous outputs are generated. But, at the same time, code coverage and predicate evaluation may include too much irrelevant information that could hamper their discriminative power. This is what motivates DS-PROXIMITY and SD-PROXIMITY, namely, to improve the discriminative power of failure signatures by pruning fault-irrelevant information.

- **Lesson 3: No free lunch.** While the above experiments show that DS-PROXIMITY and SD-PROXIMITY are better than CC-PROXIMITY and PE-PROXIMITY and CC-PROXIMITY and PE-PROXIMITY are in turn superior to FP-PROXIMITY and ST-PROXIMITY, one cannot take such superiority as absolute because many other factors should be considered besides proximity goodness. In the first place, more sophisticated fingerprinting is accompanied by higher overhead. For example, dynamic slicing may incur an average of 60 times slowdown when running on an expensive runtime instrumentation engine like Valgrind, while only minimum instrumentation is needed to collect failure points and stack traces. Besides runtime overhead, the applicability is also a concern. For instance, fault localization-based approaches (e.g., SD-PROXIMITY) generally require passing executions to fingerprint failures into likely fault locations, so, when no passing executions are available, these approaches may simply be not applicable. Therefore, nothing valuable comes for free and many factors should be considered in practice.

### 6 Discussions

#### 6.1 Related Work

Failure indexing is a critical component in bug tracking systems [42]. A bug tracking system supports bug diagnosis and software evolution by keeping records of reported failures. With proper support of failure indexing, bug tracking systems can facilitate duplicate removal, failure prioritization, and parallel debugging [43], all of which are essential for timely and efficient quality control.

Some bug tracking systems, like Bugzilla [44], are designed for manual failure reporting. Software developers or technically savvy people report encountered failures by manually typing in critical failure attributes, e.g., the platform, the failure stack trace, the submitter-perceived severity, etc. By storing the reported information into databases, failure indexing on the provided attributes is automatically supported. For example, one can easily retrieve all failures that manifest on FreeBSD with a severity level of five. Recently, automated approaches have been...
developed to remove duplicates and assign failures to the appropriate developers based on human beings’ descriptions of failures [45], [46], [47], [48], [49]. We did not compare with these techniques because they are mainly based on static text descriptions of failures, whereas this paper investigates how to extract failure signatures from program dynamic data without human intervention.

In order to save users’ hassles in failure reporting, some bug tracking systems collect program failures from production runs and index them based on failure venues [1], [2], [50], [51]. Given that these systems have done a great job in indexing crashing failures, this paper focuses on noncrashing failures that are becoming increasingly dominant [7].

Several researchers have previously investigated the problem of indexing noncrashing failures, although not under the name of failure indexing. Dickinson et al. propose a technique called cluster filtering to assist developers in finding failing traces from a set of mostly passing executions [36], [52]. Later on, Podgurski et al. report a study on clustering failure reports [10]. Francis et al. further improve the study of software failure analysis by utilizing tree-based methods for both failure clustering and failure classification [53]. In comparison, we propose here a failure indexing framework which incorporates existing and our newly proposed techniques into a failure indexing model and provides a set of evaluation metrics for fair comparison. Therefore, this paper is orthogonal to studies like [53] because each technique results in a proximity matrix on which clustering and classification algorithms can be applied.

We believe in the necessity of this failure indexing framework because scientific studies demand problem formulation and evaluation metrics. The effect of this kind of research can be extrapolated from the influence of Reniers’ thesis work [27], [54], where he formalizes the fault localization problem and proposes the PDG-based T-score to quantify localization accuracy. As a result, fault localization research becomes more principled and quantitative advancements are reported in following studies [24], [29], [30], [34], [55]. We hope this paper can affect the study of failure proximity in a similar way.

In this study, we demonstrate the possibility of utilizing fault localization algorithms to extract fault-relevant failure signatures. This method of analysis differs from most previous analysis, e.g., [10], [11], [36], [52], [53], [56], [57], [58], [59], [60], in that these previous studies do not consider fault localization algorithms as an effective way of extracting fault-relevant failure signatures. In particular, it is worth mentioning that, although [59], [60] employ sophisticated statistical methods (co-clustering and latent topic models, respectively) to cluster failures, neither of them employ fault localization techniques or show the intent to do so. This study clearly demonstrates the utility of fault localization algorithms beyond locating software faults. Given the abundance of fault localization algorithms [23], [24], [27], [28], [29], [30], [34], [55], [61], we expect many failure analysis tasks can be reexamined by utilizing the power of numerous fault localization algorithms.

An important downstream application of failure indexing is parallel debugging [43]: Once failures are divided into several clusters, multiple developers can debug each cluster of failures in parallel. In [43], Jones et al. score the quality of a failure clustering result by measuring the cost of clearing all of the bugs if all failure clusters are debugged in parallel. An interesting finding related to this study is that, under this measurement, it does not matter much if either likely fault location or execution profile is used as the signature. We note that this does not contradict the finding of this paper because [43] compares the clustering quality through the debugging cost, while this paper studies the quality of failure proximity. Very likely, in [43], the debugging cost-based evaluation metric, together with the particular clustering algorithm chosen by the authors, has masked the difference in different signatures. An in-depth study of how failure proximity interacts with different clustering algorithms and how the interaction affects downstream applications of failure clustering (e.g., parallel debugging and duplicate removal) is extremely valuable and it constitutes part of our future work.

This study also relates to dynamic program slicing. Dynamic slicing [26], [62] is a debugging technique that captures the executed statements that are involved in the computation of an erroneous value. It is used in this paper as a method to extract failure signatures. To the best of our knowledge, this is the first attempt at studying the effectiveness of dynamic slices in failure proximity. There are multiple variants of dynamic slices [63], [64] and studying their usage in failure indexing is on our research agenda. In [12], Orso et al. classify data dependence into different categories and, thus, dynamic slices can be computed in multiple steps. We are interested in investigating the effect of different types of data dependence on DS-PROXIMITY in our future work.

Finally, this work also relates to the analysis of rank data [65]. In practice, many kinds of data, especially those involving opinions and judgments, are represented as rank data. In this study, we fingerprint failures into predicate rankings and this is the first time failures are represented as rank data. Consequently, some interesting questions can be explored. For example, we expect more accurate fault localization can be achieved by aggregating individual rankings. Furthermore, this work proposes a weighted form of Kendall’s tau distance to accommodate the speciality of predicate rankings. In general, the weighted Kendall’s tau distance can be applied to any rank data where entity weights need to be considered.

6.2 Threats to Validity

Like any empirical studies, threats to validity should be considered in interpreting the experimental results. First, because the six faults used in this study are either “subclause-missing” or “off-by-one” errors, threats to internal validity may concern about the particular fault types. We chose “subclause-missing” and “off-by-one” errors because they are frequently committed by developers and are usually the root cause of noncrashing failures. However, as none of the proximity techniques examined in this study depend on fault types, the study result is not confined to “subclause-missing” and “off-by-one” errors. We demonstrate a spectrum of techniques from which a user can choose whichever he/she believes to be the most effective. Moreover, with the indexing framework laid down, each can propose and evaluate new techniques that he/she comes up with.

Another threat to internal validity is the assumption that the culprit for each failure is one and only one fault. But, in reality, a failure could be caused by multiple faults and it is...
highly debatable how to decide if a failure is caused by one fault or multiple faults given the fact that more than one fault could be active in the failing execution. Further investigation and research on this problem are needed in the future.

Second, because our experiments are based on the six faults in three mid-sized programs, threats to external validity may exist when experimental results are generalized to arbitrary program failures. For this reason, more case studies with larger programs that contain more than two faults should be carried out in the future. While some more case studies can definitely help, they can hardly save the world because of the unbounded ways in which various programs fail. The properties of different proximities and lessons we learned are based on reasoning with strong support from experiments. Therefore, the discussion and results will likely hold beyond the three subject programs. We especially expect them to help guide further investigations.

Finally, threats to construct validity concern the appropriateness of the metrics we used for effectiveness evaluation. We proposed the Silhouette Coefficient and the $A$-score to quantify the goodness of failure proximity and the accuracy of failure assignment. Although the two metrics may not agree with human judgment in all aspects, they do capture major ingredients that human beings care about. We intentionally avoid subjectiveness in our metric design so that all techniques can be objectively compared. Ultimately, all techniques need to be subject to human evaluation in practice when in-house research becomes mature enough.

7 CONCLUSION

In this paper, we presented the first systematic study of failure proximity, a research problem critical to many failure analysis tasks but, unfortunately, not yet formally studied. We formulated the problem, examined six representative techniques, and carefully studied their properties. We subjected the six techniques to three mid-sized programs and objectively evaluated their effectiveness and properties. We found that proximities that are good for crashing failures do not work well for noncrashing failures; existing noncrashing failure proximities based on code coverage and predicate evaluation deliver better but not quite satisfactory results; the two new proximities proposed by us, which are based upon statistical fault localization and dynamic slicing, respectively, supersede the other four proximities in terms of the proximity quality but may incur high overhead. In summary, we believe that the proposed framework will help researchers explore more effective techniques and, also, we hope the proposed metrics will promote quantifiable advancements in the future.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation under Grants ITR CCR-0325603, IIS-08-42769, BDI-05-15813, CNS-0708464, and CNS-0720516. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.