WDD: Weighted Delta Debugging

Xintong Zhou*, Zhenyang Xu*, Mengxiao Zhang*, Yongqiang Tian†, and Chengnian Sun*
*School of Computer Science, University of Waterloo, Waterloo, Canada
Emails: x27zhou@uwaterloo.ca, zhenyang.xu@uwaterloo.ca, m492zhan@uwaterloo.ca, cnsun@uwaterloo.ca
†Department of Computer Science and Engineering,
The Hong Kong University of Science and Technology, Hong Kong, China
Email: yqtian@ust.hk

***Abstract*—Delta Debugging is a widely used family of algorithms (*e.g.*, ddmin and ProbDD) to automatically minimize bug-triggering test inputs, thus to facilitate debugging. It takes a list of elements with each element representing a fragment of the test input, systematically partitions the list at different granularities, identifies and deletes bug-irrelevant partitions.**

Prior delta debugging algorithms assume there are no differences among the elements in the list, and thus treat them uniformly during partitioning. However, in practice, this assumption usually does not hold, because the size (referred to as weight) of the fragment represented by each element can vary significantly. For example, a single element representing 50% of the test input is much more likely to be bug-relevant than elements representing only 1%. This assumption inevitably impairs the efficiency or even effectiveness of these delta debugging algorithms.

This paper proposes Weighted Delta Debugging (WDD), a novel concept to help prior delta debugging algorithms overcome the limitation mentioned above. The key insight of WDD is to assign each element in the list a weight according to its size, and distinguish different elements based on their weights during partitioning. We designed two new minimization algorithms, W_{ddmin} and W_{ProbDD}, by applying WDD to ddmin and ProbDD respectively. We extensively evaluated W_{ddmin} and W_{ProbDD} in two representative applications, HDD and Perses, on 62 benchmarks across two languages. On average, with W_{ddmin}, HDD and Perses took 51.31% and 7.47% less time to generate 9.12% and 0.96% smaller results than with ddmin, respectively. With W_{ProbDD}, HDD and Perses used 11.98% and 9.72% less time to generate 13.40% and 2.20% smaller results than with ProbDD, respectively. The results strongly demonstrate the value of WDD. We firmly believe that WDD opens up a new dimension to improve test input minimization techniques.

***Index Terms*—Test Input Minimization, Delta Debugging, Program Reduction**

I. Introduction

A bug-triggering test input, which causes a program to fail, often contains many bug-irrelevant elements. These elements usually complicate the use of the test input to debug the program. Test input minimization is a technique that automatically minimizes the size of the input by removing the irrelevant elements while keeping the failure-inducing parts. It helps developers to focus on the essential parts of the input that cause the failure. Many minimization techniques [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] have been proposed and widely used in various scenarios [11], [12], [13], [14], [15], especially in facilitating software testing and debugging [16], [17], [18].

Delta Debugging [1] is a widely used family of algorithms to automatically minimize bug-triggering test inputs. Typically, delta debugging algorithms take a test input as a list of elements, with each element representing a fragment of the test input (*e.g.*, a token, a line, or a tree node). Then it partitions the list into sets of elements (referred to as partitions) at different granularities, systematically identifies and deletes partitions that are bug-irrelevant. State-of-the-art algorithms in this family include Minimizing Delta Debugging (ddmin) [1] and Probabilistic Delta Debugging (ProbDD) [19]. The first delta debugging algorithm ddmin systematically minimizes the list of elements in a binary-search style. The generality, effectiveness, and efficiency of ddmin make it a fundamental minimization algorithm in many subsequently proposed minimization tools [5], [7]. The other algorithm, ProbDD [19] is a recently proposed variant of ddmin. It improves the efficiency of ddmin by leveraging a probabilistic model to guide the minimization process.

In practice, delta debugging algorithms are often applied to the tree representations of the inputs rather than plain lists of tokens or lines to achieve better minimization performance, *a.k.a.*, tree-based minimization. For example, Hierarchical Delta Debugging (HDD) proposed by Misherghi and Su [5] represents the input as a tree structure (*e.g.*, a parse tree), and then uses ddmin to minimize each level of the tree from coarse to fine. Another example is Perses [7], a minimization technique that further improves HDD by leveraging context-free grammar to ensure the syntactic validity during minimization. Perses applies ddmin on the child node list of quantified nodes (*i.e.*, a type of nodes whose children are independent to each other in terms of syntax validity) in the parse tree. Both HDD and Perses show significant superiority in handling structured inputs compared to directly applying delta debugging to the flat list representations of the inputs.

Limitations. One significant limitation of prior delta debugging algorithms [1], [19] is that they overlooked the effect of element size in minimization, and thus the efficiency or even effectiveness of minimization is impaired. Specifically, ddmin performs a binary-search style deletion and iteratively divides the list into smaller partitions evenly by length (*i.e.*, the number of elements). However, due to the varying sizes of elements, ddmin fails to achieve the true evenness[1] and generates partitions with significantly different sizes. For

[1]The true evenness indicates that the size of each partition approximately equals to each other. The size of a partition is normally measured by the number of tokens it contains, that is to say, the number of tokens in each partition is approximately equal.

example, when HDD invokes ddmin to minimize the bug-triggering input of LLVM-19595 [20], the largest and smallest partitions produced by a partitioning operation can contain 8,752 and 5 tokens, respectively. However, ddmin treats these uneven partitions equally, neglecting an important statistical observation, *i.e.*, *larger partitions are more likely to contain the failure-inducing elements and thus less likely to be removed*. As a result, ddmin spends significant efforts in removing large but unlikely to be removed elements during the minimization process, which restricts its performance in large and complex bug-triggering inputs. As for ProbDD, while it successfully refines the partitioning strategy of ddmin with its probabilistic model, it still lacks awareness of the varying sizes of elements during partitioning, thus leading to suboptimal performance. More details of this limitation and its affect is illustrated in §III.

Weighted Delta Debugging. In this paper, we propose Weighted Delta Debugging (WDD), a novel concept to improve prior delta debugging algorithms by overcoming the aforementioned limitation. The key insight of WDD is to take the sizes of elements into consideration and assign each element a weight based on its size. By doing so, WDD can perform a more rational weight-based partitioning strategy, thereby enhancing minimization performance. We apply WDD to two representative delta debugging algorithms, ddmin and ProbDD, and propose two new algorithms, W_{ddmin} and W_{ProbDD}, respectively. At a high level, W_{ddmin} improves ddmin by performing a weighted binary-search style minimization, while W_{ProbDD} enhances ProbDD by incorporating the weights of elements as a new factor into the probabilistic model which guides the partitioning.

We extensively evaluate W_{ddmin} and W_{ProbDD} on 62 benchmarks across two languages, *i.e.*, C and XML, by substituting them for ddmin and ProbDD, respectively, in two application scenarios, HDD [5] and Perses [7]. The results demonstrate that W_{ddmin} and W_{ProbDD} significantly outperform ddmin and ProbDD in efficiency and effectiveness, respectively. On average, after substituting W_{ddmin} for ddmin, HDD and Perses use 51.31% and 7.47% less time to produce 9.12% and 0.96% smaller results, respectively. Moreover, with W_{ProbDD}, HDD and Perses obtain 13.40% and 2.20% smaller results with 11.98% and 9.72% less time than using ProbDD, respectively.

Contribution. This paper makes the following contributions.

- We present Weighted Delta Debugging (WDD), a novel concept that helps prior delta debugging algorithms overcome the limitation of being unaware of the different sizes among the elements in the input list.
- We realize WDD in two representative delta debugging algorithms, ddmin and ProbDD, and propose two new algorithms, W_{ddmin} and W_{ProbDD}, respectively.
- We comprehensively evaluate W_{ddmin} and W_{ProbDD} on 62 benchmarks in different application scenarios. The results demonstrate the superiority of W_{ddmin} and W_{ProbDD} over ddmin and ProbDD, respectively, thus highlighting the significance of WDD in improving test input minimization.
- For replication, we make the artifacts of this paper publicly available [21]. We also release the source code of W_{ddmin} and W_{ProbDD} in the Perses [22] repository for further research and applications.

II. Background

Test input minimization facilitates the software debugging process by automatically minimizing the size of the bug-triggering test input. This technique is highly demanded as it helps developers to focus on the essential parts of the test input and saves the time and effort required to identify the root cause of the bug. For example, both GCC [23] and LLVM [24] have explicitly announced that the bug-triggering program should be minimized before being reported. Test input minimization also assists many other software engineering tasks, such as program analysis [13] and slicing [15].

To facilitate presentation, we introduce the notations below,

- $\mathbb{E}$ denotes the set of all possible elements in test inputs
- l denotes a test input, which is a list of elements with elements drawn from $\mathbb{E}$
- $\mathbb{L}$ denotes the universe of possible test inputs, namely, $l \in \mathbb{L}$.
- $\mathbb{B} = \{\mathrm{T}, \mathrm{F}\}$ where T for true and F for false.
- $\psi : \mathbb{L} \to \mathbb{B}$ is a property test function returning T if the given input preserves a certain property, F otherwise.
- $w : \mathbb{E} \to \mathbb{N}$ is a weight function computing the weight (a natural number, such as 0, 1, and 2) of an element.

With these symbols, the problem of test input minimization can be formalized as follows.

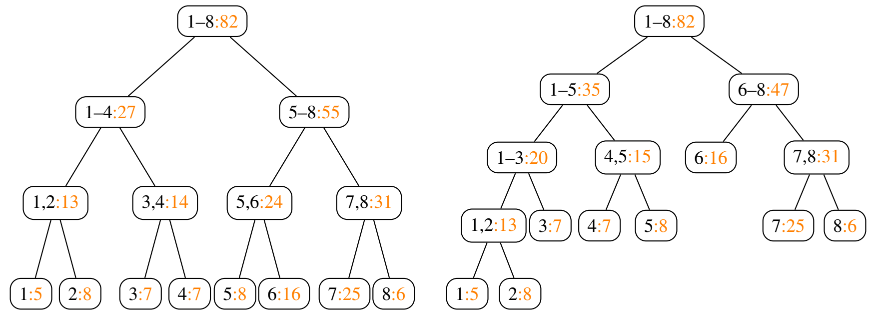
Definition II.1 (Test Input Minimization)**.** *Given a test input $l \in \mathbb{L}$ for a program and a property ψ exhibited by l, e.g., triggering a bug or generating an unexpected output when the program executes with l, the objective of test input minimization is to produce a test input $l_{min} \in \mathbb{L}$ that has a minimal number of elements and still exhibits ψ, i.e., $\psi(l_{min}) = \mathrm{T}$.*

Many techniques [1], [5], [19], [7], [6], [25], [26] have been proposed to automate test input minimization. Delta debugging algorithms, *e.g.*, ddmin and ProbDD, are among the most general and widely used techniques, upon which many advanced tools such as HDD and Perses are built. Since our approaches, *i.e.*, W_{ddmin} and W_{ProbDD}, are the improved versions of ddmin and ProbDD, respectively, we first explain the workflows of ddmin and ProbDD with an example.

Fig. 1(a) displays a program that triggers a real-world compiler bug GCC-71626 [27]. It triggers GCC to crash when compiling the program. We aim to minimize this program to the smallest size while still triggering the compiler bug, thus facilitating debugging. Taking the program as plain text and performing delta debugging algorithms on it directly is inefficient, as the program is highly structured. In practice, delta debugging is usually wrapped in tree-based techniques, *e.g.*, HDD and Perses, being applied on the tree level. In the tree representation of the program, *e.g.*, the parse tree, there are eight nodes at the same level right under the root node (highlighted in orange in Fig. 1(a)), each corresponding to a distinct part of the program such as a `typedef` statement, or a

```
1 typedef long long llong; ...........① w = 5
2 test2char64(char *p) {} ............② w = 8
3 test1char8(char c) {} ..............③ w = 7
4 test1short32(short c) {} ...........④ w = 7
5 test2short32(short *p) {} ..........⑤ w = 8
6 typedef llong vllong1 \
  __attribute__(( \
  __vector_size__(sizeof(llong)))); ..⑥ w = 16
7 vllong1 test2llong1(llong *p) {
      llong c = *test1char8;
      vllong1 v = {c};
      return v;
  } ..................................⑦ w = 25
8 int main() {} ......................⑧ w = 6
```

(a) A program that triggers GCC to crash.



(b) The search space of ddmin. (c) The search space of W_{ddmin}.

Fig. 1: A motivating example. In each subfigure, the weights of the nodes or the partitions are highlighted in orange.

function definition. To minimize the program, both HDD and Perses invoke ddmin or ProbDD to minimize the tree nodes starting from this level, *i.e.*, $[1, 2, 3, 4, 5, 6, 7, 8]$.

A. Workflow of ddmin

Given l and ψ, ddmin works in the following steps.

Step 1: Split l into n partitions evenly by length. For each partition p, test if p alone preserves ψ, *i.e.*, $\psi(p) = \text{T}$. If yes, remove all other partitions from l and resume Step 1 with $n = 2$; otherwise, go to Step 2.

Step 2: Test if the complement of each partition p preserves ψ, *i.e.*, $\psi(l \setminus p) = \text{T}$. If yes, remove p from l and resume Step 1 with $n = n - 1$; otherwise, go to Step 3.

Step 3: Terminate if each partition p contains only one element; otherwise, double n and resume Step 1.

Starting from $n = 2$ and following the above steps, ddmin performs 30 property tests in total to minimize the program in Fig. 1(a). The specific property tests ddmin performs during the minimization process are shown in Fig. 2(a). Note that ddmin may produce duplicate test inputs, which are not listed in the figure, since in practice they can be recognized and skipped by caching the tests that have been performed [1], [4].

B. Workflow of ProbDD

Different from ddmin, which follows a predefined pattern to perform the deletion operations, ProbDD [19] employs a probabilistic model to guide the entire minimization process. The key insight of ProbDD is to estimate the probability of each element appearing in the minimized result with a probabilistic model. Given l and ψ, and a map *probs* that stores the estimated probabilities of each element in l appearing in the minimized result (the initial probability of each element is set to a same value, *e.g.*, 0.2), ProbDD works in the following steps.

Step 1: Sort the elements in l in ascending order of their probabilities. Select a prefix *pre* from the sorted list that maximizes the expectation of the number of elements that can be removed, *i.e.*, $|pre| \times \prod_{e \in pre}(1 - probs[e])$.

Step 2: Test if the complement of *pre* preserves ψ, *i.e.*, $\psi(l \setminus pre) = \text{T}$. If yes, remove *pre* from l, set the probabilities of the elements in *pre* to 0, and go to Step 4; if not, go to Step 3.

Step 3: Increase the probabilities of elements in *pre* according to the probabilistic model [19], then go to Step 4.

Step 4: Terminate if the probabilities of all the elements in l reach 1; otherwise, go to Step 1.

Following the above steps, the minimization process of the example program in Fig. 1(a) is shown in Fig. 2(c). Each property test is represented with two rows, where the first row displays the elements selected (complement of *pre*) for testing, and the second row shows the probability of each element after the test. The selected elements and the updated probabilities are highlighted with blue and yellow, respectively. Starting from the same initial probability (set to 0.2 in this case), ProbDD performs 15 property tests to finish the minimization process.

C. 1-Minimality

The ultimate goal of test input minimization is to obtain the globally minimal result, where no smaller input can exhibit ψ. However, previous work has proven that obtaining the global minimality is NP-complete [1], [5]. In practice, the goal is usually relaxed to local minima. First presented by DD [1], 1-minimality has been widely adopted by a series of works [5], [7], [3], [28] as the criterion of minimality evaluation. A minimized input is considered 1-minimal if no single element can be further removed without losing the property ψ. HDD [5] extends the principle of 1-minimality to tree structures, introducing 1-tree-minimality, which promising that, in the tree representation of the input, no single tree node can be further removed without violating the property. To achieve 1-tree-minimality, tree-based techniques, *e.g.*, HDD [5] and Perses [7], typically operate in a *fixpoint mode*. In this mode, the minimization process is repeatedly applied to the minimized result until no more tree nodes can be removed from the result.

III. Motivation

As Fig. 1(a) shows, the code snippets represented by different nodes vary in size. For example, while node ① represents a typedef statement containing 5 tokens, node ⑦ defines the function test2llong1 with 25 tokens. This discrepancy in size of nodes can affect the efficiency and effectiveness of minimization. However, both ddmin and ProbDD fail to capture this information and treat all nodes uniformly, thus leaving room for improvement. This is where out concept of WDD

	Inputs for Property Tests								ψ
1	1	2	3	4	5	6	7	8	F
2	1	2	3	4	5	6	7	8	F
3	1	2	3	4	5	6	7	8	F
4	1	2	3	4	5	6	7	8	F
5	1	2	3	4	5	6	7	8	F
6	1	2	3	4	5	6	7	8	F
7	1	2	3	4	5	6	7	8	F
8	1	2	3	4	5	6	7	8	F
9	1	2	3	4	5	6	7	8	F
10	1	2	3	4	5	6	7	8	F
11	1	2	3	4	5	6	7	8	F
12	1	2	3	4	5	6	7	8	F
13	1	2	3	4	5	6	7	8	F
14	1	2	3	4	5	6	7	8	F
15	1	2	3	4	5	6	7	8	F
16	1	2	3	4	5	6	7	8	F
17	1	2	3	4	5	6	7	8	F
18	1	2	3	4	5	6	7	8	F
19	1	2	3	4	5	6	7	8	F
20	1	2	3	4	5	6	7	8	**T**
21	1		3	4	5	6	7	8	F
22	1		3	4	5	6	7	8	**T**
23	1		3		5	6	7	8	F
24	1		3		5	6	7	8	F
25	1		3		5	6	7	8	**T**
26	1		3			6	7	8	F
27	1		3			6	7	8	F
28	1		3			6	7	8	F
29	1		3			6	7	8	F
30	1		3			6	7	8	F

(a) ddmin

	Inputs for Property Tests								ψ
1	1	2	3	4	5	6	7	8	F
2	1	2	3	4	5	6	7	8	F
3	1	2	3	4	5	6	7	8	F
4	1	2	3	4	5	6	7	8	F
5	1	2	3	4	5	6	7	8	F
6	1	2	3	4	5	6	7	8	F
7	1	2	3	4	5	6	7	8	F
8	1	2	3	4	5	6	7	8	**T**
9	1	2	3			6	7	8	F
10	1	2	3			6	7	8	F
11	1	2	3			6	7	8	F
12	1	2	3			6	7	8	F
13	1	2	3			6	7	8	F
14	1	2	3			6	7	8	F
15	1	2	3			6	7	8	F
16	1	2	3			6	7	8	F
17	1	2	3			6	7	8	F
18	1	2	3			6	7	8	F
19	1	2	3			6	7	8	F
20	1	2	3			6	7	8	F
21	1	2	3			6	7	8	F
22	1	2	3			6	7	8	**T**
23	1		3			6	7	8	F
24	1		3			6	7	8	F
25	1		3			6	7	8	F
26	1		3			6	7	8	F

(b) W_{ddmin}

	Inputs for Property Tests								ψ
1	1	2	3	4	5	6	7	8	F
	0.2	0.3	0.3	0.2	0.2	0.3	0.3	0.3	
2	1	2	3	4	5	6	7	8	F
	0.41	0.3	0.3	0.41	0.41	0.3	0.3	0.3	
3	1	2	3	4	5	6	7	8	F
	0.41	0.3	0.46	0.41	0.41	0.46	0.3	0.46	
4	1	2	3	4	5	6	7	8	F
	0.41	0.59	0.46	0.41	0.41	0.46	0.59	0.46	
5	1	2	3	4	5	6	7	8	F
	0.63	0.59	0.46	0.41	0.63	0.46	0.59	0.46	
6	1	2	3	4	5	6	7	8	F
	0.63	0.59	0.67	0.6	0.63	0.46	0.59	0.46	
7	1	2	3	4	5	6	7	8	F
	0.63	0.59	0.67	0.6	0.63	0.65	0.59	0.65	
8	1	2	3	4	5	6	7	8	F
	0.63	0.59	0.67	0.6	0.63	0.65	1.00	0.65	
9	1	2	3	4	5	6	7	8	**T**
	0.63	0	0.67	0.6	0.63	0.65	1.00	0.65	
10	1		3	4	5	6	7	8	**T**
	0.63		0.67	0	0.63	0.65	1.00	0.65	
11	1		3		5	6	7	8	F
	1.00		0.67		0.63	0.65	1.00	0.65	
12	1		3		5	6	7	8	**T**
	1.00		0.67		0	0.65	1.00	0.65	
13	1		3			6	7	8	F
	1.00		0.67			1.00	1.00	0.65	
14	1		3			6	7	8	F
	1.00		0.67			1.00	1.00	1.00	
15	1		3			6	7	8	F
	1.00		1.00			1.00	1.00	1.00	

(c) ProbDD

	Inputs for Property Tests								ψ
1	1	2	3	4	5	6	7	8	F
	0.2	0.2	0.2	0.2	0.2	0.56	0.56	0.2	
2	1	2	3	4	5	6	7	8	F
	0.2	0.2	0.2	0.2	0.2	0.56	1.00	0.2	
3	1	2	3	4	5	6	7	8	F
	0.2	0.28	0.2	0.2	0.28	0.78	1.00	0.2	
4	1	2	3	4	5	6	7	8	F
	0.2	0.42	0.3	0.3	0.42	0.78	1.00	0.2	
5	1	2	3	4	5	6	7	8	F
	0.2	0.42	0.49	0.49	0.42	0.78	1.00	0.33	
6	1	2	3	4	5	6	7	8	**T**
	0.2	0	0.49	0.49	0	0.78	1.00	0.33	
7	1		3	4		6	7	8	F
	0.43		0.49	0.49		0.78	1.00	0.71	
8	1		3	4		6	7	8	F
	1.00		0.66	0.66		1.00	1.00	0.71	
9	1		3	4		6	7	8	F
	1.00		1.00	0.66		1.00	1.00	0.71	
10	1		3	4		6	7	8	**T**
	1.00		1.00	0		1.00	1.00	0.71	
11	1		3			6	7	8	F
	1.00		1.00			1.00	1.00	1.00	

(d) W_{ProbDD}

Fig. 2: The detailed minimization process of ddmin, W_{ddmin}, ProbDD, and W_{ProbDD}. The elements selected for the property test in each iteration are highlighted in blue, with the leftmost column indicating the index of each property test. In Fig. 2(c) and Fig. 2(d), the probabilities updated after each test are highlighted in yellow. The last column of each figure shows the result of the property test ψ. In this case, all the four algorithms minimize the input list to the same result, which is $[1, 3, 6, 7, 8]$.

comes into play. The key insight of WDD is to assign each element a weight that matches its size, and perform weight-based partitioning. We first define the weight of elements in delta debugging, based on which, we present two new delta debugging algorithms, W_{ddmin} and W_{ProbDD}, by applying WDD to ddmin and ProbDD, respectively.

Definition III.1 (Weight)**.** *The weight of an element in the input list of delta debugging is defined as the size of the fragment represented by the element. The weight of a partition is the sum of the weights of all elements in the partition. The size is typically measured by the number of tokens.*

A. Improving ddmin

Fig. 1(b) visualizes the search space of ddmin in a tree, illustrating that ddmin splits the list evenly by length to conduct a binary search-style deletion. However, it fails to achieve the true evenness due to the effect of different weights of nodes. As highlighted in orange in Fig. 1(b), the weights of partitions on each level vary significantly, which can impair the efficiency of ddmin. That is because, statistically speaking, a larger partition is more likely to contain the failure-inducing elements, and thus less likely to be removed. However, ddmin fails to capture this information and handles all nodes equally, leading to its efficiency being hampered by spending a large amount of attempts on deleting nodes that are unlikely to be successfully removed. For instance, the largest node (node ⑦) in the previous example, which is the core element to trigger the compiler bug, is attempted to be removed from the list with partitions for 13 times during the minimization.

Different from ddmin, W_{ddmin} considers the weights of elements and performs a weight-based partitioning to make the actual size of each partition as close as possible. The search space of W_{ddmin} based on this strategy is shown in Fig. 1(c). Following this search space, W_{ddmin} finish the minimization of the example program with only 26 property tests, and attempts to remove node ⑦ only 12 times. The detailed minimization process is shown in Fig. 2(b). This improvement is much more significant for larger and more complex inputs, as demonstrated in §VI-B2.

B. Improving ProbDD

As described in §II-B, ProbDD strives to maximize the expectation of the number of elements that can be removed during partitioning. However, the number of elements does not necessarily correspond to the number of tokens that can be deleted. For example, given two elements with the same probability of being removed, the one with more tokens (*i.e.*, larger weight) should be chosen to remove first, since deleting it contributes more to global minimization process. The performance of ProbDD is suboptimal since it fails to consider the weight of elements when constructing the probabilistic model. To fill this gap, W_{ProbDD} leverages the weight information of elements to refine the probabilistic model of ProbDD, and uses this model to guide partitioning. As shown in Fig. 2(d), boosted by the weighted model, W_{ProbDD}

minimizes the example program with only 11 property tests. It is worth clarifying that, although in this example, ProbDD and $\mathrm{W_{ProbDD}}$ produce the same minimized result, our evaluation in §VI demonstrates the superior effectiveness of $\mathrm{W_{ProbDD}}$ over ProbDD in practice by producing smaller minimized results.

IV. Weighted Minimizing Delta Debugging

This section describes the application of WDD to improve the efficiency of ddmin. Algorithm 1 details $\mathrm{W_{ddmin}}$, with our extensions beyond ddmin highlighted with grey blocks. Compared to ddmin, $\mathrm{W_{ddmin}}$ has a different partitioning strategy weightedPartition on line 20, and an additional deletion pass ensureOneMinimal on line 28 to ensure 1-minimality.

Started with the whole input l as the only partition (line 2), $\mathrm{W_{ddmin}}$ performs systematic deletion operations on the partitions and their complements, and iteratively splits the partitions into smaller ones. If a partition *ptn* alone preserves the property (*i.e.*, $\psi(ptn)$ on line 8), all the other partitions are removed, and the algorithm restarts with this single remaining partition (line 7-11). If the complement of a partition exhibits the property (*i.e.*, $\psi(complement)$ on line 14), $\mathrm{W_{ddmin}}$ removes the partition and restarts with the remaining partitions (line 12-17). If no partition or complement exhibits ψ, $\mathrm{W_{ddmin}}$ calls weightedPartition (line 18) to split the partitions into smaller ones based on the weights of the elements in these partitions, and then start a new iteration. This process terminates when the partition list *partitions* is empty (line 6). Then $\mathrm{W_{ddmin}}$ performs an additional deletion pass by calling ensureOneMinimal (line 4) to make sure the produced result is 1-minimal.

Algorithm 1: Weighted Minimizing Delta Debugging

```
Input: l ∈ 𝕃: the input list of elements.
Input: w : 𝔼 → N: the weights of each element.
Input: ψ : 𝕃 → 𝔹: the property to be preserved.
Output: the minimized list that preserves the property.
1  l_min ← l
2  partitions ← [l]
3  l_min ← wddRec(partitions, l_min, ψ, w)
4  return ensureOneMinimal(l_min, ψ)
5  Function wddRec(partitions, l_min, ψ, w):
6      while |partitions| ≠ 0 do
7          foreach ptn ∈ partitions do
8              if ψ(ptn) then
9                  l_min ← ptn
10                 partitions ← weightedPartition([ptn], w)
11                 return wddRec(partitions, l_min, ψ, w)
12         foreach ptn ∈ partitions do
13             complement ← l_min \ ptn
14             if ψ(complement) then
15                 l_min ← complement
16                 partitions ← partitions \ [ptn]
17                 return wddRec(partitions, l_min, ψ, w)
18         partitions ← weightedPartition(partitions, w)
19     return l_min
20 Function weightedPartition(partitions, w):
21     result ← []
22     foreach ptn ∈ partitions do
23         if |ptn| = 1 then continue // skip this partition
24         halfSum ← 0.5 × Σ_{e∈ptn} w(e)
25         p1, p2 ← split ptn into two partitions with weight sum of
             each close to halfSum
26         result ← result + [p1, p2] // add p1, p2 to result
27     return result
28 Function ensureOneMinimal(l_min, ψ):
29     loopStart: foreach element ∈ l_min do
30         complement ← l_min \ [element]
31         if ψ(complement) then
32             l_min ← complement
33             goto loopStart
34     return l_min
```

A. Weighted Partitioning Strategy

The main extension of $\mathrm{W_{ddmin}}$ is the partitioning strategy, as shown in function weightedPartition (line 20-27). Unlike ddmin, which partitions the input list l evenly by the *number* of elements, $\mathrm{W_{ddmin}}$ aims to split l evenly by the *weight* of elements, striving to *make the weight of each partition as close as possible*. (line 24-26). Notably, if a partition from the current iteration contains only one element, the partition will be excluded from the partition list in the next iteration (line 23), because the partition cannot be further divided.

Revisiting the example in Fig. 1(a), by applying the weight-based partitioning strategy, the search space is reorganized as shown in Fig. 1(c). While the tree is not balanced in terms of the number of elements, it achieves balance for the weight of each partition. Quantitatively, $\mathrm{W_{ddmin}}$ strives to minimize the standard deviation of the weights of partitions during partitioning. For example, in the second iteration (corresponding to the third level of the tree in Fig. 1(b) and Fig. 1(c)), the standard deviation of the partition weights of ddmin (*i.e.*, $[13, 14, 24, 31]$) is 7.43, whereas that of $\mathrm{W_{ddmin}}$ (*i.e.*, $[20, 15, 16, 31]$) is only 6.34.

B. 1-Minimality of W_{ddmin}

$\mathrm{W_{ddmin}}$ guarantees 1-minimality with an additional deletion pass, as shown in function ensureOneMinimal (line 28-34). Because of the weight-based partitioning strategy, larger elements are isolated earlier in the deletion process. For example, in Fig. 1(c), node ⑥ is isolated as a separate partition in the third iteration, and it cannot be removed in the current iteration. However, in practice, the deletion of some nodes may benefit the deletion of other nodes [1], [5], [6]. To ensure 1-minimality, $\mathrm{W_{ddmin}}$ attempts to remove each remaining element individually in the end by calling function ensureOneMinimal (line 4). The loop (line 29-33) iteratively checks whether each remaining element can be removed without losing the property. If so, the element is removed and the loop restarts. This process continues until no element can be further removed, so that 1-minimality is guaranteed.

C. Time Complexity of W_{ddmin}

$\mathrm{W_{ddmin}}$ does not shrink or enlarge the search space of ddmin. Instead, $\mathrm{W_{ddmin}}$ follows the similar deletion process as ddmin with a more rational partitioning strategy. Therefore, by design,

W_{ddmin} has the same worst-case time complexity as ddmin, *i.e.*, $O(n^2)$ [1], where n is the number of elements in the input list.

Average Time Complexity. We argue that W_{ddmin} can achieve higher overall efficiency than ddmin in practice. The key insight of W_{ddmin} is that the probability of an element being removed varies with its weight, and there is a negative correlation between them. Intuitively, an element with a larger weight, *i.e.*, representing a larger fragment of a test input, is less likely to be removed than a smaller one, as it is more likely to contain the failure-inducing elements. The statistical validation of this observation is provided in §VI-A. With this insight, we expect that W_{ddmin} can achieve better efficiency than ddmin. We perform a simulation below to demonstrate this.

D. Synthetic Analysis for Average Time Complexity

The inherent complexity of delta debugging problem prevents us from proving W_{ddmin} is better than ddmin in all cases, which is also not necessarily true in practice. Therefore, we design this simulation to compare the efficiency of W_{ddmin} and ddmin.

1) Analysis Setup: First, we randomly synthesize a set of lists and predetermine their minimization results. Next, we perform W_{ddmin} and ddmin on the lists and record the numbers of property tests required by each algorithm on each list respectively. The minimization results are predetermined based on the probability of each element being removed, and the probabilities are calculated based on the assumption below.

Assumption IV.1 (Randomness)**.** *For a random input, each token has the same probability of being removed.*

Given this assumption and the probability of a token being removed p_0, the probability of an element with w tokens being removed p_e equals to p_0^w. That is because an element can be removed only if all its tokens can be removed. With the input lists of elements synthesized randomly, this assumption helps quantitatively distinguish the probabilities of elements with different weights being removed, so that we can predetermine the minimization result. This assumption is not necessary for the correctness of W_{ddmin} in practice.

With the above assumption, we perform the simulation as follows. To synthesize a random input list, we first generate a length n of the list, where n is a random integer between 2 and 1,000 (*i.e.*, $n \in [2, 1000]$), and the total number of tokens represented by the elements in the list, which is a random integer between n and $10n$. The number of tokens for each element is distributed randomly, for instance, a list of length 4 with 10 tokens could be $[1,\ 3,\ 2,\ 4]$. To predetermine the minimization result, we first generate a random value $p_0 \in (0, 1)$, which represents the probability of each token being removed. Then, we calculate the probability of each element being removed p_e based on assumption IV.1. After that, we generate a random value $p \in [0, 1]$ for each element, and compare it with p_e to determine whether the element can be removed. The element can be removed if $p < p_e$, otherwise, it cannot be removed. Based on the established result, the property ψ is preserved if all the non-removable elements are included in the list. We execute W_{ddmin} and ddmin to minimize the synthesized list, and record their numbers of property tests during the minimization process, respectively. The effect of randomness is eliminated by repeating the single process for a large number of times. Specifically, we perform ddmin and W_{ddmin} on 5,000 randomly synthesized lists.

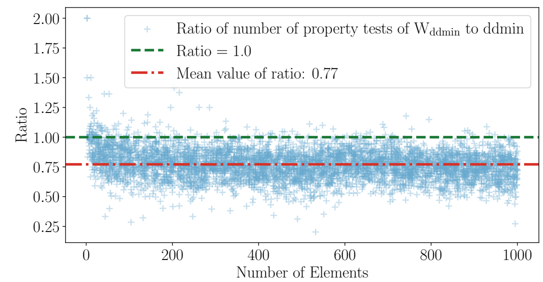


Fig. 3: The simulation results of W_{ddmin} and ddmin on the synthetic data.

2) Analysis Result: The detailed results are shown in Fig. 3. On average, W_{ddmin} uses 23% fewer property tests than ddmin to finish the minimization. The results emulatively demonstrate the superior efficiency of W_{ddmin} compared to ddmin in the ideal case where the probabilities of elements being removed are negatively correlated with their weights. We verify this correlation and evaluate the practical efficiency of W_{ddmin} on real-world benchmarks in §VI-B.

V. Weighted Probabilistic Delta Debugging

To demonstrate the generality of WDD, we applied the concept of WDD to improve ProbDD (a representative variant of ddmin) and thus proposed a new minimization algorithm W_{ProbDD}. As described in §II-B, with the model that tracks the expected probability of each element remaining in the result, the partitioning principle of ProbDD is to prioritize the deletion of elements with lowest probability and maximize the expected number of elements that can be successfully removed. However, the ultimate goal of the minimization is to delete the most tokens possible, instead of the most elements. Due to the different sizes of elements, there is a gap between the principle of ProbDD and the ultimate goal of the minimization, which makes ProbDD suboptimal. To bridge this gap, W_{ProbDD} improves ProbDD by incorporating the weight of elements as a new factor into the probabilistic model.

Algorithm 2 shows the workflow of W_{ProbDD}, and the key extensions beyond ProbDD are highlighted with grey blocks. When deciding the partition to remove in each test (implemented in function $\mathrm{getPartitionToRemove}$), the fundamental principle of W_{ProbDD} is to *(1) prioritize the deletion of elements that are likely to remove larger weight*, and *(2) maximize the expected value of weight that can be removed.* To realize the principle, W_{ProbDD} first sorts the elements in the list in descending order, by the expectation of the value of weight that can be removed by attempting to delete the element. This value equals to the product of the probability of the element can be removed and the value of its weight (line 10).

Algorithm 2: Weighted Probabilistic Delta Debugging

```
Input: l ∈ 𝕃: the input list of elements.
Input: w : 𝔼 → N: the weights of each element.
Input: ψ : 𝕃 → 𝔹: the property to be preserved.
Input: p0: the initial probability for each element.
Output: the minimized list that preserves ψ.
1  l_min ← l
2  probs ← {n → p0 | n ∈ l} // the probability function that
       records and returns the probability of each element in l
3  while not shouldTerminate(probs) do
4      ptn ← getPartitionToRemove(l_min, probs, w)
5      complement ← l_min \ ptn
6      if ψ(complement) then l_min ← complement
7      else probs ← updateProbs(ptn, probs)
8  return l_min
9  Function getPartitionToRemove(l_min, probs, w):
10     l_sorted ← sort the elements in l_min by the value of
         w(element) * (1 − probs(element)) in descending order
11     result ← [], ptn ← [], gain_max ← 0
12     foreach element ∈ l_sorted do
13         ptn ← ptn + [element]
14         weight ← Σ_{n_i ∈ ptn} w(n_i)
15         probOfDeletion ← Π_{n_j ∈ ptn} (1 − probs(n_j))
16         gain ← weight × probOfDeletion
17         if gain > gain_max then
18             gain_max ← gain
19             result ← ptn
20     return result
21 Function shouldTerminate(probs):
       // Implementation skipped. Same as ProbDD in [19].
22 Function updateProbs(ptn, probs):
       // Implementation skipped. Same as ProbDD in [19].
```

After that, W_{ProbDD} determines the partition to remove in each test with the sorted list. Technically, starting from the first element in the sorted list, W_{ProbDD} can include any number of elements in the partition to remove in the next test. While including more elements increases value of weight that can be removed, it also decreases the probability of the test passing. To balance the trade-off, W_{ProbDD} chooses the number of elements for removal that maximizes the expectation of the value of weight that can be removed successfully. To this end, W_{ProbDD} redefines the gain function in ProbDD with the weights of elements as $Gain(m) = \sum_{i=1}^{m} w_i \cdot \prod_{j=1}^{m}(1 - p_j)$ where m is the number of elements to be removed, w_i is the weight of the i-th selected element, and p_j is the probability of being remained of the j-th element. As shown in function getPartitionToRemove (line 11-20), W_{ProbDD} selects a certain prefix of the sorted list l_{sorted} that maximizes the gain function as the partition, and attempts to remove this partition in the next test. The complexity of this process is $O(n)$, where n is the length of the list.

The rest steps of W_{ProbDD} are similar to ProbDD, including performing property tests, and updating the probabilities of elements according to prior test results. We exclude the explanation of these steps here, instead, and refer the readers to the original paper of ProbDD [19] for details.

A. Minimality of W_{ProbDD}

W_{ProbDD} promises the same minimality as ProbDD, which is conditional 1-minimality. The result of ProbDD is 1-minimal under the assumption that the deletability of each element is independent. However, this assumption typically does not hold in practice, since the deletion of some elements may affect the deletability of other elements. For example, even if a statement that defines a variable is bug-irrelevant, it can only be removed after all the statements that use the variable are removed. Despite sharing the same minimality, W_{ProbDD} is expected to generate smaller results than ProbDD, since W_{ProbDD} always strives to maximize the weight (*i.e.*, the number of tokens) that can be removed in the next test. We evaluate the practical effectiveness of W_{ProbDD} in §VI-B.

B. Time Complexity of W_{ProbDD}

W_{ProbDD} shares the same worst-case time complexity as ProbDD, which is $O(n)$ [19], where n is the length of the input list. In practice, the deletion strategy of W_{ProbDD} *i.e.*, maximizing the expected weight can be removed, not only enhances effectiveness, but also speeds up the minimization process. That is because, successfully removing a partition containing a large number of tokens can usually make the execution of subsequent tests faster. Therefore, we expect that W_{ProbDD} can outperform ProbDD in terms of time efficiency. This expectation can hardly be verified by a simulation, so we directly evaluate the efficiency of W_{ProbDD} on real benchmarks in §VI-B.

VI. Evaluation

In this section, we verify the significance of WDD by evaluating the effectiveness and efficiency of W_{ddmin} and W_{ProbDD}. we explore to what extent W_{ddmin} and W_{ProbDD} outperform ddmin and ProbDD in different application scenarios, respectively. We select HDD and Perses for evaluation as they are two state-of-the-art test input minimization tools that rely on delta debugging. For each of the two techniques, we implement W_{ddmin} and W_{ProbDD} versions to replace their original versions with ddmin and ProbDD, respectively, and compare their performance with the original versions. For ease of presentation, we refer to HDD with ddmin, W_{ddmin}, ProbDD and W_{ProbDD} as HDD_{d}, HDD_{w}, HDD_{p}, and HDD_{wp}, respectively. Similarly, the four versions of Perses are referred to as Perses_{d}, Perses_{w}, Perses_{p}, and $\text{Perses}_{\text{wp}}$, respectively. All the minimization techniques for evaluation are executed in the fixpoint mode as described in §II-C. For fair comparison, all experiments were conducted on an Ubuntu 22.04 server with an Intel Xeon CPU @ 2.60GHz and 512 GB RAM, using a single-process, single-threaded.

We aim to answer the following research questions.

1) What is the correlation between element weight and the probability of being removed in practice?
2) How does the performance of W_{ddmin} compare to ddmin?
3) How does the performance of W_{ProbDD} compare to ProbDD?

Benchmarks. We conducted experiments with 62 benchmarks. Each benchmark triggers a real-world bug in a certain language processor, and is considerably large and complex, aligning with real-world application scenarios of test input minimization. Specifically, we utilized the following benchmarks.

- **C**: We collected 32 C programs from previous studies [7], [9], [4]. These programs trigger real bugs in LLVM and GCC, and are large, complex with 77,723 tokens on average.
- **XML**: To increase the diversity of the benchmark suite, we included 30 XML files, with each triggering a bug in Basex [29], a widely used XML database and Xquery processor. These benchmarks are also large and complex, containing 20,197 tokens on average.

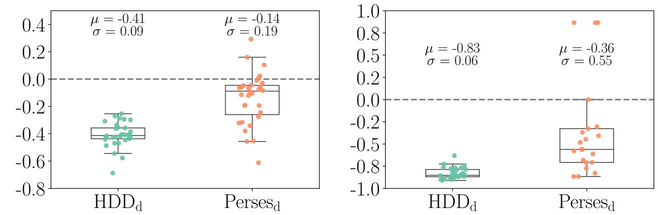
Metrics. We used the following metrics to evaluate different algorithms, following [7], [3], [4], [9], [19].

- **S(#)**: the number of tokens in the minimized result. A lower value means a more effective minimization by removing more property-irrelevant elements.
- **T(s)**: the processing time in seconds. Shorter time means higher efficiency.
- **Speed**: the number of tokens deleted per second. Using processing time to gauge efficiency is not comprehensive, for cases where one approach generates a smaller result but also takes longer time. We measure the number of tokens deleted per second to balance the trade-off between effectiveness and time consumption.
- **Wilcoxon signed-rank test** [30]: to measure the statistical significance of the improvements our our approaches. A small p-value (typically < 0.05) from this test suggests a statistically significant difference between the paired data.

A. *RQ1: Element Weight* v.s. *Deletion Probability Correlation*

The first question we are curious about is the correlation between the weights of elements and their probabilities of being removed. Since the fundamental observation behind W_{ddmin} is that larger elements are less likely to be deleted than smaller ones, we would like to verify if our assumption, *i.e.*, *the probability of elements being deleted is negatively correlated with their weights*, holds during the execution of ddmin in practice. Specifically, for an input list l and its minimized result l_{min}, the probability of elements with weight w being deleted $P_{del}(w)$ is defined as the ratio of the number of elements with weight w that are deleted to the total number of elements with weight w, *i.e.*, $P_{del}(w) = \frac{\#(w,l) - \#(w,l_{min})}{\#(w,l)}$, where $\#(w,l)$ denotes the number of elements with weight w in list l. To evaluate the correlation, we calculate the Spearman's rank correlation coefficient [31] between the probabilities of elements being deleted and their weights for each execution of ddmin. Being widely used in practice [32], [33], Spearman's rank correlation coefficient ρ [31] is a non-parametric measure of the strength and direction of association between two ranked variables. The value of ρ ranges from -1 to 1: $\rho = 1$ indicates a perfect positive correlation, $\rho = -1$ indicates a perfect negative correlation, and $\rho = 0$ implies no correlation.

To answer this research question, we use HDD_d and $Perses_d$ to minimize the test inputs in our benchmarks, and record the weights of elements before and after each execution of ddmin. Cases where no elements are removed are excluded, since ρ is undefined in these scenarios. We then calculate ρ for each execution of ddmin. Since ddmin is normally performed multiple times when minimizing a test input, the ρ of each benchmark is calculated as the average of the ρ values from all executions of ddmin for that benchmark.



(a) C Programs (b) XML inputs

Fig. 4: The Spearman correlation coefficient ρ between the weights of elements and their probabilities being deleted in ddmin. Each data point represents the mean of the ρ values of all ddmin executions on a benchmark.

As shown in Fig. 4, overall, in each scenario of HDD_d and $Perses_d$, and for both C programs and XML inputs, our assumption is preserved. As shown in Fig. 4, in the four scenarios, only 5 cases of C programs and 3 cases of XML inputs in $Perses_d$ have ρ values greater than 0, while all other cases have ρ values less or equal to 0. Specifically, when minimizing the C programs with HDD_d and $Perses_d$, the mean ρ values are -0.41 and -0.14, respectively. For the XML inputs, the mean ρ values are -0.83 and -0.36, respectively. Although the ρ values vary across different applications and benchmarks, all are less than 0, indicating a negative correlation between the probability of elements being deleted and their weights in ddmin executions, thus validating our assumption.

RQ1: The probability of elements being deleted is negatively correlated with their weights in ddmin executions in both HDD and Perses, to varying degrees. This validation provides a solid foundation for the design of W_{ddmin}.

B. *W_{ddmin}* v.s. *ddmin*

For this question, we compare the performance of HDD_w and $Perses_w$ with HDD_d and $Perses_d$, respectively. The detailed results are shown in Table I.

1) Effectiveness: Overall, W_{ddmin} is more effective than ddmin in both HDD and Perses. On average, HDD_w generates 7.81% and 16.51% smaller results than HDD_d on C and XML benchmarks, respectively, with a p-value of 5.06×10^{-5} overall. In Perses, the results of $Perses_w$ are 1.04% and 0.27% smaller than those of $Perses_d$ on C and XML benchmarks, respectively, with a p-value of 0.53 overall.

Notably, while the above results demonstrate the superior effectiveness of W_{ddmin} over ddmin, the improvement of W_{ddmin} over ddmin in Perses is not as significant as that in HDD. Given the different design of HDD and Perses, this result is expected. Unlike HDD that fully relies on ddmin to perform tree node deletion, Perses customizes different deletion strategies for

TABLE I: Results of all algorithms in HDD and Perses on all benchmarks. Better results in each pair are highlighted in bold.

Benchmark	HDD_d		HDD_w		$Perses_d$		$Perses_w$		HDD_p		HDD_{wp}		$Perses_p$		$Perses_{wp}$	
	T(s)	S(#)	T(s)	S(#)	T(s)	S(#)	T(s)	S(#)	T(s)	S(#)	T(s)	S(#)	T(s)	S(#)	T(s)	S(#)
clang-18596	15,664	452	**9,882**	**414**	4,985	260	**4,885**	260	7,337	516	**6,371**	**482**	4,966	261	**4,664**	**260**
clang-19595	11,332	285	**7,973**	**260**	4,073	156	**3,977**	156	6,554	310	**5,787**	**244**	4,311	156	**4,162**	156
clang-20680	24,517	377	**13,289**	**366**	19,090	533	**12,727**	**525**	11,748	522	**9,153**	**338**	20,864	**532**	**7,317**	537
clang-21467	28,907	455	**12,845**	**415**	6,240	177	**6,197**	177	7,898	423	**6,872**	**310**	6,162	**169**	**5,951**	177
clang-21582	25,012	1,197	**13,257**	**999**	7,291	559	**7,156**	559	6,499	1,112	**4,892**	**1,089**	5,770	626	**5,475**	**559**
clang-22337	42,489	**299**	**16,023**	323	2,986	236	**2,757**	236	2,176	338	**2,090**	**301**	1,679	263	**1,648**	**250**
clang-22382	11,601	**182**	**5,332**	194	887	144	**879**	144	893	**197**	**791**	200	473	**142**	**460**	144
clang-22704	49,870	**95**	**31,873**	97	5,627	78	**5,155**	78	6,987	122	**5,268**	**90**	**2,915**	78	5,427	**72**
clang-23309	54,142	1,085	**23,356**	**1,075**	**2,995**	475	3,266	475	4,962	1,136	**4,249**	**1,089**	1,701	**457**	**1,506**	483
clang-23353	61,023	**144**	**38,212**	153	3,868	98	**3,005**	98	3,252	**185**	**2,814**	218	1,261	149	**1,240**	**98**
clang-25900	39,481	478	**10,486**	**346**	2,690	252	**2,248**	252	2,312	518	**2,206**	**471**	1,014	**238**	**970**	252
clang-26350	72,997	391	**35,172**	**369**	9,494	189	**9,418**	189	12,165	599	**11,673**	**455**	5,395	**232**	**5,046**	241
clang-26760	37,413	321	**11,487**	**229**	4,543	91	**3,989**	91	3,678	306	**2,901**	**276**	2,129	112	**1,922**	**90**
clang-27137	207,071	582	**74,903**	**545**	14,322	268	**13,094**	268	20,225	**638**	**18,724**	661	**7,616**	**196**	7,992	268
clang-27747	4,468	299	**2,684**	**244**	2,160	117	**1,464**	117	1,493	324	**1,035**	**302**	1,187	**137**	**1,027**	150
clang-31259	16,977	**556**	**9,863**	594	3,134	384	**2,964**	384	**3,967**	576	5,307	**571**	2,229	393	**2,200**	**384**
gcc-58731	8,369	431	**5,614**	**375**	3,389	213	**3,325**	213	4,807	416	**3,428**	**368**	3,112	237	**2,779**	**213**
gcc-59903	25,965	**545**	**12,006**	734	4,815	497	**4,231**	**382**	**3,729**	771	5,217	**410**	3,634	381	**3,020**	**300**
gcc-60116	24,708	1,281	**13,077**	**1,137**	**3,177**	443	3,252	443	5,323	1,245	**4,083**	**889**	2,196	428	**2,092**	**404**
gcc-60452	40,082	491	**13,883**	**442**	2,544	350	**2,412**	350	3,559	824	**2,164**	**495**	1,676	**346**	**1,656**	350
gcc-61047	13,086	**495**	**5,803**	505	1,082	266	**1,024**	266	**1,686**	564	2,180	**512**	**796**	267	799	**266**
gcc-61383	22,599	579	**10,207**	**421**	**3,070**	**271**	3,187	274	3,524	**509**	**3,505**	514	2,768	282	**2,759**	**274**
gcc-61917	27,399	**293**	**12,531**	311	2,414	142	**2,032**	142	2,862	**327**	**2,494**	300	1,845	145	**1,386**	**142**
gcc-64990	65,997	**325**	**36,885**	378	5,216	239	**4,403**	239	9,047	601	**8,388**	**467**	3,376	239	**3,342**	239
gcc-65383	27,974	246	**12,296**	**217**	1,593	153	**1,420**	153	2,273	281	**1,800**	**275**	1,139	153	**1,113**	153
gcc-66186	22,124	605	**10,091**	**508**	2,939	327	**2,864**	327	6,688	591	**6,681**	**503**	2,691	327	**2,603**	327
gcc-66375	78,050	1,053	**28,445**	**740**	4,114	440	**3,987**	440	**10,635**	842	11,403	**813**	3,345	440	**3,105**	440
gcc-66412	32,524	491	**13,276**	**442**	2,642	350	**2,421**	350	3,301	769	**2,278**	**495**	1,794	350	**1,686**	350
gcc-66691	18,230	1,076	**11,607**	**1,022**	3,671	746	**3,553**	**602**	**3,787**	**959**	5,625	1,039	3,256	689	**3,184**	**603**
gcc-70127	73,476	617	**30,417**	**576**	4,511	301	**4,279**	301	16,367	660	**11,445**	**635**	3,406	301	**3,234**	301
gcc-70586	66,155	**792**	**43,507**	793	**6,998**	**197**	7,779	367	13,478	921	**11,363**	**763**	5,812	**168**	**5,394**	197
gcc-71626	1,742	53	**409**	53	**50**	51	53	51	146	53	**104**	53	**45**	51	46	51
Mean	39,108	518	**18,022**	**477**	4,582	281	**4,169**	**278**	6,042	567	**5,384**	**488**	3,455	280	**2,975**	**273**
xml-1	732	33	**488**	**24**	**259**	16	262	16	1,248	43	**528**	**24**	425	16	**419**	16
xml-2	1,798	60	**283**	**15**	89	15	**75**	15	1,935	51	**283**	**15**	139	15	**134**	15
xml-3	765	36	**277**	**15**	**81**	15	82	15	487	23	**236**	**15**	**115**	15	128	15
xml-4	2,545	78	**2,509**	78	1,159	13	**1,143**	13	2,197	78	**2,151**	78	1,234	13	**1,229**	13
xml-5	865	33	**242**	**15**	**86**	15	89	15	273	20	**150**	**15**	69	15	69	15
xml-6	4,100	120	**4,024**	120	667	30	**659**	30	2,688	67	**1,289**	**30**	1,145	30	**1,119**	30
xml-7	1,837	69	**1,787**	69	640	33	**629**	33	2,295	69	2,295	69	**1,062**	33	1,080	33
xml-8	4,302	138	**429**	**24**	388	16	**378**	16	2,615	78	**531**	**24**	**539**	16	546	16
xml-9	**2,182**	63	2,216	63	1,616	37	**1,591**	37	**1,756**	**68**	2,272	90	**1,446**	37	1,469	37
xml-10	2,681	102	**2,651**	102	1,411	37	**1,105**	37	1,993	102	**1,937**	102	1,155	37	**999**	37
xml-11	1,909	90	**435**	**24**	378	16	**386**	16	759	37	**450**	**24**	471	16	**465**	16
xml-12	1,637	96	**1,384**	**87**	462	16	**439**	16	2,029	91	**1,872**	**87**	752	16	**712**	16
xml-13	1,359	78	**1,332**	78	485	25	**483**	25	1,589	78	**1,519**	78	**718**	25	739	25
xml-14	4,034	153	**3,955**	153	**1,551**	43	1,553	43	4,882	160	**4,716**	**153**	**2,267**	43	2,292	43
xml-15	2,333	108	**1,012**	**51**	545	16	**534**	16	1,581	85	**993**	**51**	621	16	**620**	16
xml-16	1,738	60	**513**	**24**	437	16	**430**	16	1,157	38	**532**	**24**	**579**	16	602	16
xml-17	2,519	87	**292**	**15**	101	15	**97**	15	1,458	58	**212**	**15**	**108**	15	124	15
xml-18	785	39	**761**	39	433	16	**423**	16	1,496	50	**916**	**39**	569	16	**549**	16
xml-19	**1,512**	54	1,539	54	**628**	36	632	36	1,858	60	**1,807**	**54**	**937**	36	968	36
xml-20	2,492	99	**2,380**	99	2,117	64	**2,108**	64	**3,120**	99	3,209	99	3,502	64	**3,393**	64
xml-21	3,362	93	**3,256**	**90**	1,550	46	**1,463**	46	4,244	91	**4,192**	**90**	**2,480**	46	2,549	46
xml-22	1,454	61	**1,402**	61	346	24	**335**	24	1,633	61	**1,467**	61	480	24	**420**	24
xml-23	7,317	189	**5,905**	**165**	2,332	54	**2,183**	**51**	**6,725**	**152**	6,727	165	**3,184**	**47**	3,432	51
xml-24	**2,269**	96	2,365	96	2,092	70	**2,055**	70	2,896	96	**2,803**	96	3,421	70	**3,196**	70
xml-25	**3,114**	135	3,696	**132**	1,956	55	**1,924**	55	2,672	139	**2,042**	**135**	1,745	55	**1,710**	55
xml-26	6,238	126	**6,166**	126	**2,688**	64	2,762	64	6,373	128	**6,268**	138	4,278	64	**4,179**	64
xml-27	8,955	195	**8,769**	195	**4,382**	102	4,442	102	9,340	188	**8,529**	**186**	7,135	102	**7,030**	102
xml-28	7,690	159	**7,333**	159	**3,811**	97	3,820	97	7,818	159	**7,203**	159	5,928	97	**5,825**	97
xml-29	5,860	147	**5,173**	**138**	1,906	48	**1,861**	48	3,810	142	**3,515**	**138**	1,819	48	**1,797**	48
xml-30	6,190	147	**6,054**	147	**4,258**	78	4,261	78	7,197	147	**6,583**	147	6,807	78	**6,591**	78
Mean	3,152	98	**2,621**	**82**	1,295	38	**1,273**	38	3,004	89	**2,574**	**80**	1,838	37	**1,813**	37

different types of nodes. In Perses, ddmin is only used to minimize the list of nodes under a quantified node [7]. That is to say, compared to HDD, the deletion operations performed by ddmin (or W_{ddmin}) constitute a smaller proportion of the total operations in Perses. Therefore, improvements to the effectiveness of ddmin have a relatively moderate impact on the overall effectiveness of Perses. Moreover, the nodes under a quantified node in Perses are syntactically independent from each other [7], making the minimization less challenging. Thus, ddmin can generate results comparable to W_{ddmin}.

2) Efficiency: We first evaluate efficiency with processing time, for which W_{ddmin} outperforms ddmin in both HDD and

Perses. On average, HDD_w takes 53.92% and 16.86% less time than HDD_d to finish minimizing the C programs and XML inputs, respectively, with a p-value of 5.3×10^{-11} overall. Similarly, $Perses_w$ reduces the processing time of $Perses_d$ by 9.01% and 1.67% on C and XML benchmarks, respectively, with a p-value of 2.33×10^{-6} overall. Furthermore, considering the number of tokens deleted per second (referred as *tokens/s*) as an additional metric, while HDD_d and $Perses_d$ deletes 6.59 and 38.14 *tokens/s*, respectively, HDD_w and $Perses_w$ deletes 14.2 and 40.2 *tokens/s*, which are 115.57% and 5.40% more than those of HDD_d and $Perses_d$, respectively. These results strongly indicate that W_{ddmin} is more efficient than ddmin.

Similar with the improvement of effectiveness, while W_{ddmin} consistently achieves higher efficiency than ddmin in both HDD and Perses, the improvement is more significant in HDD than in Perses. The reason is the same as that explained in §VI-B1 for effectiveness. Moreover, the high efficiency of W_{ddmin} is based on the assumption that the probability of elements being deleted is negatively correlated with their weights, which is validated in §VI-A. In fact, the degree of this correlation can affect the efficiency of W_{ddmin}. As shown in Fig. 4, the ρ values of $Perses_d$ are generally larger than those of HDD_d, indicating a weaker negative correlation. Thus, the improvement of W_{ddmin} over ddmin in Perses is not as significant as that in HDD.

RQ2: W_{ddmin} outperforms ddmin in both effectiveness and efficiency in HDD, by generating 9.12% smaller results in 51.31% less time on average. In Perses, W_{ddmin} exceeds ddmin in efficiecny by taking 7.47% less time on average to generate the comparable results.

C. RQ3: W_{ProbDD} v.s. *ProbDD*

For this research question, we compare the performance of HDD_{wp} and $Perses_{wp}$ using HDD_p and $Perses_p$ as baselines, respectively. The minimization process of ProbDD contains nondeterminism since it may randomly select elements when their probabilities are the same. To mitigate the impact of such nondeterminism, we repeat each experiment for 5 times and report the average results. W_{ProbDD} largely eliminates the randomness of ProbDD by considering the weights of elements. The detailed results are shown in Table I.

Effectiveness. Overall, W_{ProbDD} is more effective than ProbDD by generating smaller results. On average, HDD_{wp} generates 13.91% and 9.74% smaller results than HDD_p for the C programs and XML inputs, respectively, with a p-value of 1.10×10^{-6}. Besides, $Perses_{wp}$ generates 2.43% smaller and 0.32% larger results than $Perses_p$ for the C programs and XML inputs, respectively, with a p-value of 0.42. There is no significant difference between the results of $Perses_{wp}$ and $Perses_p$. In fact, $Perses_{wp}$ generates 36 same results as $Perses_p$ out of 62 benchmarks, of which, 29 are from the XML inputs, because of the same reason explained in §VI-B2. Especially, for the XML inputs, only 11.9% property tests performed by $Perses_{wp}$ are from W_{ProbDD}, indicating that the effectiveness of $Perses_{wp}$ is largely determined by the inner deletion strategies of Perses, instead of W_{ProbDD}.

Efficiency. W_{ProbDD} achieves higher efficiency than ProbDD in both HDD and Perses. We first evaluate the efficiency of W_{ProbDD} with processing time. On average, HDD_{wp} shortens the processing time of HDD_p by 10.89% and 14.31% for the C programs and XML inputs, respectively, with a p-value of 1.06×10^{-6} overall. Similarly, $Perses_{wp}$ reduces the processing time of $Perses_p$ by 13.89% and 1.35% on each benchmark suite, respectively, with a p-value of 1.73×10^{-5} overall. Moreover, in terms of the number of tokens deleted per second as an additional metric, while HDD_p and $Perses_p$ deletes 15.82 and 39.95 *tokens/s*, HDD_{wp} and $Perses_{wp}$ deletes 24.03 and 40.36 *tokens/s*, which are 51.90% and 1.03% more than those of HDD_p and $Perses_p$, respectively.

RQ3: W_{ProbDD} outperforms ProbDD in both effectiveness and efficiency, by making HDD and Perses produce 13.40% and 2.20% smaller results in 11.98% and 9.72% less time on average, respectively.

VII. Discussion

A. Alternative Weight Assignment

In our implementation of WDD in ddmin and ProbDD in this paper, we utilize the number of tokens of each element as the weight. Although this assignment strategy is not 100% accurate, it achieves high efficiency and feasibility as it is *static, lightweight and generalizable*. Other weight assignment could also be considered, such as a dynamic weight assignment strategy based on runtime information, including factors like memory usage, IO operations, or execution time. However, such a dynamic weight assignment strategy may introduce additional overhead, potentially hindering the performance of minimization. Furthermore, runtime profiling techniques are typically language-specific, which may limit the generalizability of WDD. Overcoming these challenges and exploring the potential of dynamic WDD for language-specific minimization techniques presents an interesting direction for future work.

B. Limitations

The primary limitation of WDD is its applicability mainly to tree-structured inputs, where it is most effective when the weights (*i.e.*, token counts) of elements vary significantly. When the test inputs cannot be represented in a tree structure (e.g., random strings), while the concept of weight still exists, token count may not serve as an appropriate weight representation. Additionally, if the tree representation of the test input is highly balanced, WDD may offer only marginal improvement over traditional delta debugging methods. Nevertheless, given the widespread use of tree-based minimization techniques and the typically unbalanced nature of trees in real-world inputs, WDD remains essential for enhancing the performance of test input minimization in practical scenarios.

C. Threats to Validity

1) Threats to Internal Validity: The primary internal threat arises from the implementation of the evaluated techniques, including W_{ddmin}, W_{ProbDD}, and their respective baselines, as well as HDD and Perses. To mitigate this threat, we rigorously reproduced the the baseline techniques based on their descriptions in the original papers, and wrote multiple test cases to ensure the algorithms functioned as expected. Additionally, all authors of this paper participated in a thorough code review of the implementation. Prior to evaluating the full set of benchmarks, we randomly selected several cases, ran our algorithms on them, and manually verified the detailed results to confirm the accuracy of our implementations. We have also made our implementations publicly available for replication and facilitating further research.

2) Threats to External Validity: A key threat to external validity is the generalizability of WDD across different input formats or languages. Although WDD is designed to apply to all tree-structured inputs, variations in the tree characteristics of different inputs may impact its performance. To mitigate this threat, we evaluated WDD on two types of benchmarks: C and XML. The C benchmarks represent traditional programming languages, while the XML files represent structured inputs that are highly hierarchical but not programs. Our evaluation results demonstrate the superior performance of WDD across these diverse formats. To further address this threat, our future work includes expanding the evaluation of WDD to a broader range of benchmarks.

VIII. Related Work

We introduce two lines of related work.

Test Input Minimization. Delta Debugging [1] is the first systematic study that enlightens the research of test input minimization. It introduced an minimizing algorithm named ddmin to minimize failure-inducing test inputs, which has been described in §II-A. While ddmin is effective, its efficiency is not satisfactory as it follows a predefined pattern to partition and delete elements, overlooking the information of existing tests. To fix this issue, Wang et al. [19] proposed ProbDD. As explained in §II-B, ProbDD leverages a probabilistic model to guide the minimization process. However, both ddmin and ProbDD overlook the different sizes of elements in the list, leading to suboptimal performance. Contrastively, our approaches, W_{ddmin} and W_{ProbDD}, successfully distinguish different elements with their weights, and make more rationale partitioning decisions with considering weights, which significantly improves the performance of prior delta debugging algorithms. In practice, rather than being used directly to minimize test inputs, delta debugging algorithms are often integrated into tree-based minimization techniques for better performance. Two representative techniques are HDD [5] and Perses [7], which are chosen for our evaluation. HDD and Perses apply delta debugging algorithms to minimize the list of nodes in the tree. Thus their performance can be further improved by equipping our new delta debugging algorithms.

Program Reduction. Program reduction is a special case of test input minimization, where the input is a program. Since normally a program can be parsed into a syntax tree, tree-based test input minimization techniques, *e.g.*, HDD and Perses, can be directly applied to program reduction. Moreover, Xu *et al.* proposed Vulcan [3], which pushes the limit of 1-minimality by performing predefined program transformations. They further developed T-Rec [2], a fine-grained language-agnostic program reduction technique guided by lexical syntax. T-rec is demonstrated to not only achieve smaller minimization results than Vulcan, but also aids in deduplicating bug-triggering test inputs. Additionally, Zhang *et al.* proposed LPR [10], the first language-agnostic program reducer boosted by large language models. Furthermore, some program reduction techniques are specifically designed for certain languages. For example, C-Reduce [6] is specifically designed for reducing C/C++ programs. It incorporates various semantic-specific transformations to effectively minimize C/C++ programs. J-Reduce [25], [34], ddSMT [26] and JS Delta [35] are specifically designed for reducing Java bytecode, SMT-LIBv2 inputs, and JavaScript programs, respectively. Herfert *et al.* propose the Generalized Tree Reduction (GTR) technique which minimizes programs with a series of language-specific transformations generated by learning from a corpus of example data [36]. While these approaches are designed for specific languages, some of them, such as ddSMT, apply delta debugging under the hood. To this end, introducing our novel concept of WDD to these tools to further improve their performance is a promising direction for future work.

IX. Conclusion

This paper introduces Weighted Delta Debugging (WDD), a novel concept that incorporates the weight of elements into delta debugging. The key insight of WDD is to assign each element in the input list a weight, and distinguish different elements based on their weights during partitioning. We realize the concept of WDD in two representative delta debugging algorithms, ddmin and ProbDD, and propose W_{ddmin} and W_{ProbDD}, respectively. The extensive evaluation on 62 benchmarks demonstrates the superior performance of W_{ddmin} and W_{ProbDD}, in both effectiveness and efficiency, highlighting the significance of WDD in optimizing delta debugging algorithms. We firmly believe that WDD opens up a new dimension to improve test input minimization techniques.

Acknowledgments

We thank all the anonymous reviewers in ICSE'25 for their insightful feedback and comments. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under WHJIL, and CFI-JELF Project #40736.

REFERENCES

[1] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[2] Z. Xu, Y. Tian, M. Zhang, J. Zhang, P. Liu, Y. Jiang, and C. Sun, "T-rec: Fine-grained language-agnostic program reduction guided by lexical syntax," *ACM Transactions on Software Engineering and Methodology*, 2024.

[3] Z. Xu, Y. Tian, M. Zhang, G. Zhao, Y. Jiang, and C. Sun, "Pushing the limit of 1-minimality of language-agnostic program reduction," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 636–664, 2023. [Online]. Available: https://doi.org/10.1145/3586049

[4] Y. Tian, X. Zhang, Y. Dong, Z. Xu, M. Zhang, Y. Jiang, S.-C. Cheung, and C. Sun, "On the caching schemes to speed up program reduction," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, nov 2023. [Online]. Available: https://doi.org/10.1145/3617172

[5] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 142–151. [Online]. Available: https://doi.org/10.1145/1134285.1134307

[6] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[7] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 361–371. [Online]. Available: https://doi.org/10.1145/3180155.3180236

[8] J. L. Tian, M. Zhang, Z. Xu, Y. Tian, Y. Dong, and C. Sun, "Ad hoc syntax-guided program reduction," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 2137–2141. [Online]. Available: https://doi.org/10.1145/3611643.3613101

[9] M. Zhang, Z. Xu, Y. Tian, Y. Jiang, and C. Sun, "Ppr: Pairwise program reduction," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 338–349.

[10] M. Zhang, Y. Tian, Z. Xu, Y. Dong, S. H. Tan, and C. Sun, "Lpr: Large language models-aided program reduction," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 261–273.

[11] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 380–394. [Online]. Available: https://doi.org/10.1145/3243734.3243838

[12] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197–208.

[13] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, 2013, pp. 1–12.

[14] J. Regehr. (2015, Dec) Reducers are Fuzzers – EMBEDDED IN ACADEMIA. [Online]. Available: https://blog.regehr.org/archives/1284

[15] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Orbs: Language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 109–120.

[16] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[17] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[18] T. L. Wang, Y. Tian, Y. Dong, Z. Xu, and C. Sun, "Compilation consistency modulo debug information," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 146–158. [Online]. Available: https://doi.org/10.1145/3575693.3575740

[19] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 881–892. [Online]. Available: https://doi.org/10.1145/3468264.3468625

[20] L. Bugzilla. (2014) Bug 19595 - clang hangs on valid code at -o1 and above on x86_64-linux-gnu. [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=19595

[21] X. Zhou, Z. Xu, M. Zhang, Y. Tian, and C. Sun, "Artifact for "wdd: weighted delta debugging"," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14270380

[22] PLUverse. (2024) perses: Language and systems support for type-directed program transformations. https://github.com/uw-pluverse/perses. Accessed: 2024-08-02.

[23] GCC. (2020) A guide to testcase reduction. [Online]. Available: https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[24] LLVM. (2022) How to submit an llvm bug report. [Online]. Available: https://llvm.org/docs/HowToSubmitABug.html

[25] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 556–566. [Online]. Available: https://doi.org/10.1145/3338906.3338956

[26] A. Niemetz and A. Biere, "ddsmt: a delta debugger for the smt-lib v2 format," in *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*, 2013, pp. 8–9.

[27] G. Bugzilla. (2015) Bug 71626. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=71626

[28] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394. [Online]. Available: https://doi.org/10.1145/3243734.3243838

[29] BaseX Team. (2024) Basex. XML Database and XQuery Processor. [Online]. Available: https://github.com/BaseXdb/basex

[30] R. F. Woolson, "Wilcoxon signed-rank test," *Encyclopedia of Biostatistics*, vol. 8, 2005.

[31] J. H. Zar, "Spearman rank correlation," *Encyclopedia of Biostatistics*, vol. 7, 2005.

[32] N. S. Chok, "Pearson's versus spearman's and kendall's correlation coefficients for continuous data," Ph.D. dissertation, University of Pittsburgh, 2010.

[33] K. Ali Abd Al-Hameed, "Spearman's correlation coefficient in statistical analysis," *International Journal of Nonlinear Analysis and Applications*, vol. 13, no. 1, pp. 3249–3255, 2022.

[34] C. G. Kalhauge and J. Palsberg, "Logical bytecode reduction," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1003–1016. [Online]. Available: https://doi.org/10.1145/3453483.3454091

[35] JS Delta. (2017) JS Delta. [Online]. Available: https://github.com/wala/jsdelta

[36] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 861–871.