2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Toward a Better Understanding of Probabilistic Delta Debugging

Mengxiao Zhang*, Zhenyang Xu*, Yongqiang Tian†, Xinru Cheng*, and Chengnian Sun*
*School of Computer Science, University of Waterloo, Waterloo, Canada
Emails: m492zhan@uwaterloo.ca, zhenyang.xu@uwaterloo.ca, x59cheng@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*—Given a list L of elements and a property ψ that L exhibits, `ddmin` is a classic test input minimization algorithm that aims to automatically remove ψ-irrelevant elements from L. This algorithm has been widely adopted in domains such as test input minimization and software debloating. Recently, ProbDD, a variant of `ddmin`, has been proposed and achieved state-of-the-art performance. By employing Bayesian optimization, ProbDD estimates the probability of each element in L being relevant to ψ, and statistically decides which and how many elements should be deleted together each time. However, the theoretical probabilistic model of ProbDD is rather intricate, and the underlying details for the superior performance of ProbDD have not been adequately explored.**

In this paper, we conduct the first in-depth theoretical analysis of ProbDD, clarifying the trends in probability and subset size changes and simplifying the probability model. We complement this analysis with empirical experiments, including success rate analysis, ablation studies, and examinations of trade-offs and limitations, to further comprehend and demystify this state-of-the-art algorithm. Our success rate analysis reveals how ProbDD effectively addresses bottlenecks that slow down `ddmin` by skipping inefficient queries that attempt to delete complements of subsets and previously tried subsets. The ablation study illustrates that randomness in ProbDD has no significant impact on efficiency. These findings provide valuable insights for future research and applications of test input minimization algorithms.

Based on the findings above, we propose CDD, a simplified version of ProbDD, reducing the complexity in both theory and implementation. CDD assists in ① validating the correctness of our key findings, *e.g.*, that probabilities in ProbDD essentially serve as monotonically increasing counters for each element, and ② identifying the main factors that truly contribute to ProbDD's superior performance. Our comprehensive evaluations across 76 benchmarks in test input minimization and software debloating demonstrate that CDD can achieve the same performance as ProbDD, despite being much simplified.

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

I. Introduction

Delta Debugging [1] is a seminal family of algorithms designed for software debugging, among which `ddmin` stands out as a classic test input minimization (*a.k.a.*, test input reduction) algorithm. Given a list L of elements (modeling the test input) and a property ψ that L exhibits, `ddmin` aims to remove elements in L that are irrelevant to ψ, such that the resulting list is smaller than L yet still satisfies ψ. The `ddmin` algorithm plays a crucial role in software testing, debugging and maintenance [2]–[6], since compact and informative bug-triggering inputs are easier for developers to effectively identify root causes than large bug-triggering inputs with bug-irrelevant information [7]–[10].

To minimize a test input I that satisfies ψ, `ddmin` has been used in two primary manners. In the first manner, I is initially segmented into a list, denoted as L, which could be segmented based on characters, tokens, lines, *etc.* Subsequently, `ddmin` is directly applied to L [1], [11]. Alternatively, `ddmin` serves as a pivotal component within advanced, structure-aware test input minimization algorithms, including Perses [12], HDD [13], C-Reduce [14], and Chisel [15]. These algorithms leverage the inherent structures of I to expedite the minimization process or further reduce its size. Generally, these algorithms initiate by parsing I into a tree structure, such as a parse tree. They then iteratively extract a list L of tree nodes from the tree using heuristics and apply `ddmin` to L to gradually condense the tree. Both manners underscore the fundamental role of `ddmin` as the cornerstone of test input minimization.

In the past years, different variants of `ddmin` have been proposed to improve its performance [15]–[19], among which Probabilistic Delta Debugging (ProbDD) [16] is the state of the art, with notable superiority to other algorithms [1], [15]. When reducing L, ProbDD utilizes a theoretical probabilistic model based on Bayesian optimization to predict how likely every element in L is essential to preserve the property ψ, by assigning a probability to each element. ProbDD prioritizes deleting elements with lower probabilities, as such elements generally have a lower possibility of being ψ-relevant. Before each deletion attempt, an optimal subset of elements is determined by maximizing the Expected Reduction Gain.[1] If the deletion of this subset fails to preserve ψ, the probabilistic model increases the probability assigned to each element in the subset. As reported [20], aided by such a probabilistic model, ProbDD significantly outperforms `ddmin` by reducing the execution time and the query number.[2]

However, this probabilistic model in ProbDD is rather intricate, and the underlying mechanisms for its superior per-

[1]In each attempt, the Expected Reduction Gain is defined as the expected number of elements removed. Higher Expected Reduction Gain is preferred, as it indicates an expectation to delete more elements through this attempt.

[2]A query is a run of the property test ψ.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00117

formance have not been adequately studied. The original paper of ProbDD merely showed its performance numbers without deep ablation analysis on such achievements. Specifically, the following questions are important to the research field of test input minimization, but have not been answered yet.

1) What role do probabilities play in ProbDD, and can they be simplified without impacting performance?
2) What specific bottlenecks does ProbDD overcome to achieve improvement compared to `ddmin`?
3) How does randomness in ProbDD contribute to the performance improvement?
4) What are the potential limitations of ProbDD?

Gaining a deeper understanding of the state of the art, *i.e.*, ProbDD, is highly valuable for test input minimization tasks. By clarifying the intrinsic reasons behind its superiority, we can facilitate researchers to understand the essence of the probabilistic model, as well as its strengths and limitations. Such demystification, in our view, paves the way for enlightening future research and guides users to more effectively apply `ddmin` and its variants for test input minimization.

To this end, we conduct the first in-depth analysis of ProbDD, starting by theoretically simplifying its probabilistic model. In the original ProbDD, probabilities are used to calculate the Expected Reduction Gain, which is subsequently used to determine the next subset size. However, this process necessitates iterative calculations, impeding the simplification and comprehension of ProbDD. In our study, we initially establish the analytical correlation between the probability and subset size, allowing for probabilities and subset sizes to be explicitly calculated through formulas, thus eliminating the need for iterative updates. Further, through mathematical derivation, we discover that the probability and subset size can be considered nearly independent, each varying at an approximate ratio on their own. By theoretical prediction, the probability increases approximately by a factor of $\frac{1}{1-e^{-1}}$ ($\approx$ 1.582), and the subset size can be deduced by this probability, thus providing the potential for simplifying ProbDD.

Building upon our theoretical analysis, we conducted extensive evaluations of `ddmin`, ProbDD, and CDD across 76 diverse benchmarks. The experimental results confirm the correctness of our theoretical analysis, demonstrating how ProbDD addresses bottlenecks in `ddmin` by skipping inefficient queries, reveals the impact of randomness on results, and highlights the limitations of ProbDD. These findings provide valuable guidance for future research and the development of test input minimization algorithms.

Based on the aforementioned analysis, we propose Counter-Based Delta Debugging (CDD), a simplified version of ProbDD, to explain ProbDD's high performance. By replacing probabilities with counters, CDD eliminates the probability computations required by ProbDD, thus reducing theoretical and implementation complexity. Our experiments demonstrate that CDD aligns with ProbDD in both effectiveness and efficiency, which validates our previous analysis and findings.

Key Findings. Through both theoretical analysis and empirical experiments, our key findings are:

1) Through theoretical derivation, the probabilities in ProbDD essentially serve as monotonically increasing counters, and can be simplified. This suggests that the probability mechanism itself may not be a critical factor in ProbDD's superior performance.
2) The performance bottlenecks addressed by ProbDD are inefficient deletion attempts on complements of subsets and previously tried subsets, which should be considered to enhance efficiency.
3) Randomness in ProbDD has no significant impact on the performance. Test input minimization is an NP-complete problem, and randomness in ProbDD does not produce smaller results.
4) ProbDD is faster than `ddmin`, but at the cost of not guaranteeing 1-minimality.[3] The trade-off between effectiveness and efficiency is inevitable, and should be leveraged accordingly in different scenarios.

Contributions. We make the following major contributions.

- We perform the first in-depth theoretical analysis for ProbDD, the state-of-the-art algorithm in test input minimization tasks, and identify the latent correlation between the subset size and the probability of elements.
- We propose CDD, a much simplified version of ProbDD.
- We evaluate `ddmin`, ProbDD and CDD on 76 benchmarks, validating the correctness of our theoretical analysis. Additional experiments and statistical analysis on ProbDD further explain its superior performance, reveal the effectiveness of randomness, and demonstrate the limitations of ProbDD.
- To enable future research on test input minimization, we release the artifact publicly for replication [21]. Additionally, we have integrated CDD into the Perses project, available at https://github.com/uw-pluverse/perses.

Paper Organization. The remainder of the paper is structured as follows: § II introduces the symbols used in this study and detailed workflow of `ddmin` and ProbDD. § III presents our in-depth analysis on ProbDD, simplifying the model of probability and subset size. § IV describes empirical experiments and results, from which additional findings are derived. § V introduces CDD, which simplifies ProbDD based on our earlier findings while maintaining equivalent performance. § VII illustrates related work and § VIII concludes this study.

II. Preliminaries

To facilitate comprehension, Table I lists all the important symbols used in this paper. Next, this section introduces `ddmin` and ProbDD, with the running example shown in Fig. 1.

A. The `ddmin` Algorithm

The `ddmin` algorithm [1] is the first algorithm to systematically minimize a bug-triggering input to its essence, which has been widely adopted in program reduction [12]–[14], software debloating [15], [22] and test suites reduction [23], [24]. It takes the following two inputs:

[3]A list is considered to have 1-minimality if removing any single element from it results in the loss of its property.

TABLE I: The symbols used in this paper.

Symbol	Description	Symbol	Description
L	the list to minimize	s	the size of S
ψ	the property to preserve	$E(s)$	Expected Reduction Gain with the first s elements
l_i	the i-th element of L	e	Euler's number
$l_i.p$	the probability of l_i	r	the round number
v_i	a variant of L	s_r	the subset size in round r
S	a subset of L	p_r	the probability of each element in round r

```
l1:import math, sys
l2:input = sys.argv[1]
l3:a = int(input)
l4:b = math.e
l5:c = 3
l6:d = pow(b, a)
l7:c = math.log(d, b)
l8:crash(c)
```

(a) Original.

```
l1:import math, sys
l2:input = sys.argv[1]
l3:a = int(input)
l4:b = math.e
l5:c = 3
l6:d = pow(b, a)
l7:c = math.log(d, b)
l8:crash(c)
```

(b) By ddmin.

```
l1:import math, sys
l2:input = sys.argv[1]
l3:a = int(input)
l4:b = math.e
l5:c = 3
l6:d = pow(b, a)
l7:c = math.log(d, b)
l8:crash(c)
```

(c) By ProbDD.

Fig. 1: A running example in Python. Fig. 1(a) shows the original program, represented as a list of 8 elements (l_1, l_2, $\cdots$, l_8), in which l_8 (*i.e.*, `crash(c)`) triggers the crash. Fig. 1(b) and Fig. 1(c) show the minimized results by `ddmin` and ProbDD, with removed elements masked in gray. Both minimized programs still trigger the crash. Note that ProbDD cannot consistently guarantee the result in Fig. 1(c) and might produce larger results, due to its inherent randomness.

- L: a list of elements representing a bug-triggering input. For example, L can be a list of bytes, characters, lines, tokens, or parse tree nodes extracted from the bug-triggering input.
- ψ: a property that L has. Formally, ψ can be defined as a predicate that returns T if a list of elements preserves the property, F otherwise.

and returns a minimal subset of L that still preserves ψ, from which excluding any single element will make the minimal subset lose ψ. This algorithm has been widely used in practice to facilitate developers in debugging [11], [12], [14], [25]. It generally consists of the following three steps.

Initialize. Start by setting the initial subset size s to half of the input list L, *i.e.*, s=$|L|/2$.

Step 1: Minimize to Subset. Partition L into subsets of size s. For each subset S, check whether S alone satisfies ψ. If yes, keep only S and restart from Step 1 with $L = S$ and the subset size as half of the new L; otherwise, go to Step 2.

Step 2: Minimize to Complement. Partition L into subsets of size s. For each subset S, check whether the complement of S (*i.e.*, $L/S = \{e | e \in L \wedge e \notin S\}$) satisfies ψ. If yes, keep the complement of S and restart from Step 2 with $L = L/S$; otherwise, go to Step 3.

Step 3: Subdivide. If any of the remaining subsets has at least two elements and thus can be further divided, halve the subset size, *i.e.*, $s = s/2$ and go back to Step 1. If no subset can be further divided (i.e., the subset size is 1), `ddmin` terminates and returns the remaining elements as the result.

Round Number r. Note that we introduce a round number r at the second column of Table II. Within each round, the list L is divided into subsets of a fixed size, on which Step 1 and Step 2 are applied. A new round begins when no further progress can be made with the current subset size. This round number is *not explicitly* present in the original `ddmin` algorithm but exists *implicitly*. In subsequent sections, we will also use this concept to introduce and simplify the ProbDD algorithm.

Table II illustrates the step-by-step minimization process of `ddmin` with the running example in Fig. 1. Initially, the input L is [l_1, l_2, $\cdots$, l_8]. The `ddmin` algorithm iteratively generates variants by gradually decreasing the subset size from 4 to 1.

1) Round 1 (s=4). At the beginning, `ddmin` splits L into two subsets and generates two variants v_1 and v_2. However, neither of them preserves ψ.
2) Round 2 (s=2). Next, `ddmin` continues to subdivide these two subsets into smaller ones, and generates eight variants (*i.e.*, v_3, v_4, $\cdots$, v_{10}) by using these subsets and their complements. Specifically, the first four variants (v_3, v_4, v_5, v_6) are the subsets, and the next four variants (v_7, v_8, v_9, v_{10}) are the complements of these subsets. Again, none of these eight variants preserves ψ.
3) Round 3 (s=1). Finally, `ddmin` decreases subset size s from 2 to 1, and generates more variants. This time, v_{23}, which is the complement of the subset $\{l_5\}$, preserves ψ. Hence, the subset $\{l_5\}$ is permanently removed from L. Then for each of the remaining subsets $\{l_1\}$, $\{l_2\}$, $\cdots$, $\{l_8\}$, `ddmin` restarts testing the complement of each subset, *i.e.*, from v_{24} to v_{30}. However, none of these variants preserves ψ, and no subset can be further divided, so `ddmin` terminates with the variant v_{23} as the final result.

B. Probabilistic Delta Debugging (ProbDD)

Wang *et al.* [16] proposed the state-of-the-art algorithm ProbDD, significantly surpassing `ddmin` in minimizing bug-triggering programs on C compilers and benchmarks in software debloating. ProbDD employs Bayesian optimization [26] to model the minimization problem. ProbDD assigns a probability to each element in L, representing its likelihood of being essential for preserving the property ψ. At each step during the minimization process, ProbDD selects a subset of elements expected to yield the highest Expected Reduction Gain, and targets these elements in the subset for deletion. In this section, we outline ProbDD's workflow in Algorithm 1, paving the way for a deeper understanding and analysis of ProbDD.

Initialize (line 1). In L, ProbDD assigns each element an initial probability p_0 on line 1, representing the prior likelihood that each element cannot be removed.

Step 1: Select elements (line 4, line 14–24). First, ProbDD sorts the elements in L by probability in ascending order on line 14, and the order of elements with the same probability is determined randomly. Then, on line 19, it calculates the subset to be removed in the next attempt via the proposed Expected Reduction Gain $E(s)$, as shown in Equation (1), with $E(s)$ denoting the expected gain obtained via removing the *first*

TABLE II: Step-by-step outcomes from `ddmin` on the running example. In each column, a variant is generated and tested against the property ψ. These variants are sequentially generated from left to right. The first row displays the variant identifier, and the second row displays round number r and subset size s. In the following rows, the symbol "✓" denotes an element is included by a certain variant, while gray cells signify that the element have been removed. For the last row, T indicates that the variant still preserves the property ψ, whereas F indicates not.

Initial	Variants	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_{17}	v_{18}	v_{19}	v_{20}	v_{21}	v_{22}	v_{23}	v_{24}	v_{25}	v_{26}	v_{27}	v_{28}	v_{29}	v_{30}
Element	Round	$r=1$ (s=4)		$r=2$ (s=2)								$r=3$ (s=1)																			
l_1		✓		✓					✓	✓	✓	✓									✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
l_2		✓		✓					✓	✓	✓		✓							✓		✓	✓	✓	✓		✓	✓	✓	✓	✓
l_3		✓			✓			✓		✓	✓			✓						✓	✓		✓	✓	✓	✓		✓	✓	✓	✓
l_4		✓			✓			✓		✓	✓				✓					✓	✓	✓		✓	✓	✓	✓		✓	✓	✓
l_5			✓			✓		✓	✓		✓					✓				✓	✓	✓	✓								
l_6			✓			✓		✓	✓		✓						✓			✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
l_7			✓				✓	✓	✓	✓								✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
l_8			✓				✓	✓	✓	✓									✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
ψ		F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	F	F	F	F	F	F	F

s *elements* in L selected for deletion, and $l_i.p$ denoting the current probability of the i-th element in L.

$$E(s) = s \times \prod_{i=1}^{s} (1 - l_i.p) \tag{1}$$

Note that ProbDD has an invariant that the subset S chosen for deletion attempt is always the first s elements in L. Every time, the first s^* elements are selected as the optimal subset S, where s^* maximizes the Expected Reduction Gain $E(s)$, elaborated as Equation (2).

$$s^* = \arg\max_{s} E(s) \tag{2}$$

Step 2: Delete the Subset (line 5-9). If ψ is still preserved after the removal of S, ProbDD removes subset S on line 6, *i.e.*, keeps only the complement of S, and proceeds to Step 1. If ψ cannot be preserved after the removal, on lines 8 and 9, ProbDD updates the probability of each element in the subset S via Equation (3), and resumes at Step 1. It is important to note that if an element l_i has been individually deleted but failed, its probability $l_i.p$ will be set to 1, indicating that this element cannot be removed and will no longer be considered for deletion.

$$l_i.p \leftarrow \frac{l_i.p}{1 - \prod_{l \in S} (1 - l.p)} \tag{3}$$

Step 3: Check Termination (line 3). If every element either has been deleted, or possesses a probability of 1, ProbDD terminates. If not, it returns to Step 1.

Round Number r**.** Similar to the concept of rounds in `ddmin` (see Table II), ProbDD also has an *implicit* round number r, as introduced on line 2 in Algorithm 1 and the second row of Table III. During a round, the subset size is the same and every subset in L is attempted for deletion. Once the probabilities of all elements have been updated, the next round begins (*i.e.*, $r \leftarrow r + 1$ on line 11).

Table III illustrates the step-by-step results of ProbDD. Following the study of ProbDD [16], the initial probability p_0 is set to 0.25, resulting in subsets with a size of 4 as per Equation (2).

1) Round 1 (s=4). Similar to the example in the original paper of ProbDD [16], we assume ProbDD selects (l_1, l_4, l_5, l_8) to delete due to the randomness, thus resulting in the variant v_1. However, v_1 fails to exhibit ψ, leading to the probability of these selected elements being updated from 0.25 to $\frac{0.25}{1-(1-0.25)^4} \approx 0.37$, based on Equation (3). Next, the remaining elements with lower probability, *i.e.*, (l_2, l_3, l_6, l_7), are prioritized and selected for deletion, resulting in v_2. This time, the property test passes and these elements are removed.
2) Round 2 (s=2). Given that all probabilities of remaining elements become 0.37, the next subset size becomes 2. Subsequently, subset (l_1, l_5) are attempted to remove in v_3 and later subset (l_4, l_8) are attempted to remove in v_4, though no subset can be successfully removed. After these two attempts, all probabilities update to $\frac{0.37}{1-(1-0.37)^2} \approx 0.61$.
3) Round 3 (s=1). Finally, the subset size becomes 1, so each individual element is selected to remove alone. The elements l_4 and l_1 are finally removed from the final result in v_5 and v_7, respectively, while l_5 and l_8 are verified as non-removable, thus being returned as the final result.

III. Delving Deeper into Probability and Size

Beginning with this section, we will systematically present our findings. Each finding will be introduced by first stating the result, followed by the explanation. In this section, we theoretically analyze the trend of probability changes across rounds, and the approach to derive the optimal subset size.

A. On the Probability in ProbDD

Finding 1: The probability assigned to each element increases monotonically with the round number r, by a factor of approximately 1.582. Essentially, the probability for each element can be expressed as a function of r and p_0, i.e.,

$$p_r \approx 1.582^r \times p_0$$

An Illustrative Example. The running example illustrated in Table III leads to this finding. Observation reveals that after each element has been attempted for deletion once, *i.e.*, completing one round, the probabilities of all remaining elements are updated. The initial probability is 0.25; after v_2, it

Algorithm 1: `ProbDD`(L, ψ)

Input: L: a list to be minimized.
Input: $\psi : \mathbb{L} \to \mathbb{B}$: the property to be preserved by L.
Input: p_0: the initial probability given by the user.
Output: the minimized list that still exhibits the property ψ.

```
// Initialize the probability of each element with p_0
foreach l ∈ L do l.p ← p_0
/* The round number r, initially 0. r is not explicitly used
   in the original ProbDD algorithm. It is displayed for
   demonstrating ProbDD's implicit principles.            */
r ← 0
while ∃l ∈ L : l.p < 1 do
    // Select elements from L for deletion attempt.
    S ← SelectSubset(L)
    // Check if removing the subset preserves the property
    temp ← L \ S
    if ψ(temp) = T then L ← temp
    else
        // Calculate the factor to update probabilities
        factor ← 1 / (1 − ∏_{l∈S}(1 − l.p))
        // Update the probabilities of elements in the subset
        foreach l ∈ S do l.p ← factor × l.p
    if All elements' probability have been updated then
        // Move to the next round.
        r = r + 1
return L
Function SelectSubset(L):
    Input: L: a list of elements to be reduced.
    Output: The subset of elements that maximizes the Expected
            Reduction Gain.
    /* Sort L by ascending probability, with elements having
       the same probability in random order.              */
    sortedL ← RandomizeThenSort(L)
    S ← ∅
    currentMaxGain ← 0
    foreach l ∈ sortedL do
        tempSubset ← S ∪ {l}
        gain ← |tempSubset| × ∏_{l∈tempSubset}(1 − l.p)
        if gain > currentMaxGain then
            currentMaxGain ← gain
            S ← tempSubset
        else break
    return S
```

TABLE III: Step-by-step outcomes from ProbDD on the running example. Similar to Table II, round number, subset size and the details of each variants are presented. For each variant, the probability of each element is noted alongside.

Initial		Variants	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Element	Prob	Round	$r=1$ (s=4)		$r=2$ (s=2)		$r=3$ (s=1)			
l_1	0.25		0.37 ✓	0.37	0.61 ✓	0.61	✓ 0.61 ✓	0.61		
l_2	0.25		✓ 0.25							
l_3	0.25		✓ 0.25							
l_4	0.25		0.37 ✓	0.37	✓ 0.37	0.61				
l_5	0.25		0.37 ✓	0.37	0.61 ✓	0.61	✓ 0.61	1	✓ 1	✓ 1
l_6	0.25		✓ 0.25							
l_7	0.25		✓ 0.25							
l_8	0.25		0.37 ✓	0.37	✓ 0.37	0.61	✓ 0.61 ✓	0.61	✓ 0.61	1
ψ			F	T	F	F	T	F	T	F

changes to 0.37; following v_4, it increases to 0.61; and by the end of v_8, it reaches 1. Consequently, we hypothesize that with each deletion attempt, the probability approximately increases in a predictable manner. Through appropriate simplification, we can theoretically model this trend, and thereby model the entire progression of probability changes.

1) Assumption for Theoretical Analysis: Besides the above observation from a concrete example, theoretical analysis is necessary. To refine the mathematical model of ProbDD for easier representation, analysis and derivation, we assume that the number of elements in L is always divisible by the subset size. With this assumption, the probability of each element will be updated in the same manner; as a result, before and after each round, the probabilities of all elements are always the same, as shown in Lemma III.1. This assumption is often applicable in practice. For instance, in the running example in Table III, before each round, the probabilities associated with each remaining element are identical, ensuring that all subsets are of identical size. Furthermore, the probabilities of elements are updated to the same next value after the round.

Lemma III.1. *If the number of elements in L is always divisible by the subset size, then after each round, the probabilities of all elements will always remain the same.*

Proof. We use mathematical induction to prove this lemma.
Base Case. Initially, all probabilities are set to the same value. Hence, before the first round, the probabilities of all elements are identical.
Inductive Step. Assume that before a given round, the probabilities of all elements are identical (induction hypothesis). After failing to delete a subset S, ProbDD updates the probability of each element of S according to Equation (3). This formula depends solely on two factors: the current probability of each element of S, *i.e.*, $l_i.p$, and the size of the subset $|S|$. For $l_i.p$, by the induction hypothesis, all elements have the same probability at the beginning of the round; for $|S|$, if the total number of elements in L is divisible by the subset size, then every subset in the round will have the same size $|S|$. Therefore, both factors $l_i.p$ and $|S|$ are identical for all elements in a subset, and the probabilities of these elements are updated to the same new value using Equation (3). Furthermore, as all subsets undergo the same update process, the probabilities of all elements in the list will remain identical at the end of the round.
Conclusion. The probabilities of all elements remain identical at the end of this round. □

Consequently, as long as the total number of elements is always divisible by the subset size, the probabilities of all elements will remain identical throughout the process. Take the running example in Table III as a demonstration. During the reduction, the number of elements is always divisible by the subset size in each round, *i.e.*, s=4, s=2, s=1. Therefore, starting with an initial probability of 0.25, the probability of each elements remain identical after each round, being 0.37, 0.61 and 1, respectively.

While it is not always possible for the number of elements to be divisible by the subset size, the elements will still be partitioned as evenly as possible. However, such indivisibilities make the theoretical simplification of ProbDD nearly impossible. Based on our observation when running ProbDD, being slightly uneven during partitioning does not significantly

affect probability updates. Moreover, we will demonstrate that the simplified algorithm derived from this assumption has no significant difference from ProbDD in § V, via thorough experimental evaluation.

2) Probability vs. *Subset Size Correlation:* In the second step, we derive the correlation between probability and subset size. Based on the assumption in the previous step, the probability of each element is identical and represented as p_r in round r, thus the formula of Expected Reduction Gain from Equation (1) can be simplified to

$$E(s) = s \times (1 - p_r)^s \tag{4}$$

Given the probability of elements p_r in the round r, s_r can be derived through gradient-based optimization, *i.e.*, $E'(s_r) = 0$. Therefore, the optimal size s_r to maximize $E(s)$ is $-\frac{1}{\ln(1-p_r)}$. Subsequently, we can also deduce the next probability to be $p_{r+1} = \frac{p_r}{1-(1-p_r)^{s_r}}$. In summary, the correlation between probability and subset size can be simplified as Equation (5) and Equation (6), in which subset size s_r is determined by probability p_r, and probability p_{r+1} in the next round is determined by both p_r and s_r.

$$\begin{cases} s_r = -\dfrac{1}{\ln(1-p_r)} & (5) \\ p_{r+1} = \dfrac{p_r}{1-(1-p_r)^{s_r}} & (6) \end{cases}$$

3) Trend of Probability Changes: Through Equation (6), $p_{r+1} > p_r$ always holds, indicating a monotonic increase of the probability of elements. However, there is still room for simplification, as s_r can be represented by p_r, implying that p_{r+1} can be represented solely by p_r.

Lemma III.2. *p is increased by a factor $\frac{1}{1-e^{-1}}$,* i.e.*,*

$$p_r = \frac{p_{r-1}}{1-e^{-1}} = \frac{p_0}{(1-e^{-1})^r} \tag{7}$$

Proof. Given $s_r = -\frac{1}{\ln(1-p_r)}$, we can deduce that $1 - p_r = e^{-\frac{1}{s_r}}$.

Subsequently, we substitute $1 - p_r$ into Equation (6), obtaining

$$p_{r+1} = \frac{p_r}{1-(1-p_r)^{s_r}} = \frac{p_r}{1-(e^{-\frac{1}{s_r}})^{s_r}}$$
$$= \frac{p_r}{1-e^{-1}} \approx 1.582 \times p_r$$

Equivalently, the approximate probability after round r can be derived given only p_0, *i.e.*,

$$p_r = \frac{p_0}{(1-e^{-1})^r} \approx 1.582^r \times p_0$$

□

Therefore, through empirical observations on the running example, coupled with theoretical derivation and simplification, we have identified the pattern of probability changes *w.r.t.* the round number r, *i.e.*, $p_r = \frac{p_0}{(1-e^{-1})^r} \approx 1.582^r \times p_0$.

B. On the Size of Subsets in ProbDD

Finding 2: The size of subsets in r-th round can be analytically pre-determined given only the probability of this round, *i.e.*, $s_r = \arg\max_{s \in \mathbb{N}^+} s \times (1-p_r)^s$, which is either $\lfloor -\frac{1}{\ln(1-p_r)} \rfloor$ or $\lceil -\frac{1}{\ln(1-p_r)} \rceil$.

Based on the Finding 1, the probability p_r can be approximately estimated by the current round number r via a factor. Consequently, we can further derive the subset size s_r by maximizing the Expected Reduction Gain in ProbDD.

Lemma III.3. *The optimal subset size s_r in round r is either $\lfloor -\frac{1}{\ln(1-p_r)} \rfloor$ or $\lceil -\frac{1}{\ln(1-p_r)} \rceil$.*

Proof. The Expected Reduction Gain is determined by the formula $E(s_r) = s_r \times (1-p_r)^{s_r}$, which increases initially with s_r and then decreases as s_r grows further, enabling the optimal solution to be identified through derivative analysis. Therefore, we can deduce the optimal s_r by solving $E'(s_r) = 0$. Therefore, the optimal size of subsets s_r in r-th round is $-\frac{1}{\ln(1-p_r)}$, which will be rounded to either $\lfloor -\frac{1}{\ln(1-p_r)} \rfloor$ or $\lceil -\frac{1}{\ln(1-p_r)} \rceil$. The final subset size should be chosen based on which integer results in a larger Expected Reduction Gain. □

Lemma III.3 allows the subset size to be analytically pre-determined, thus providing the potential for simplification of ProbDD and leading to the proposal of CDD (detailed in § V).

IV. Empirical Experiments

In addition to the theoretical derivation above, we conduct an extensive experimental evaluation on `ddmin` and ProbDD to gain deeper insights and achieve further discoveries. Specifically, we reproduce the experiments on `ddmin` and ProbDD by Wang *et al.* [16], and then delve deeper into ProbDD, analyzing its randomness, the bottlenecks it overcomes, and its 1-minimality. Furthermore, we evaluate our proposed CDD (which will be presented in § V), validating our previous theoretical analysis. Due to limited space, we present the results of both ProbDD and CDD together within this section, but this section primarily focuses on discussing ProbDD, while the next section will focus on CDD.

A. Benchmarks

To extensively evaluate `ddmin`, ProbDD and CDD, we use the following three benchmark suites (76 benchmarks in total), covering various use scenarios of minimization algorithms.

- BM_C: 20 large bug-triggering programs in C language, each of which triggers a real-world compiler bug in either LLVM or GCC. The original size of benchmarks ranges from 4,397 tokens to 212,259 tokens. This benchmark suite has been used to evaluate test input minimization work [12], [16], [27], [28].
- BM_DBT: source programs of 10 command-line utilities. The original size of benchmarks ranges from 34,801 tokens to 163,296 tokens. This benchmark suite was collected by Heo *et al.* [15] and used to evaluate software debloating techniques [15], [29], [30].

- BM_{XML}: 46 XML inputs triggering 8 unique bugs in Basex, a widely-used XML processing tool. The original size of benchmarks ranges from 19,290 tokens to 20,750 tokens. This benchmark suite is generated via Xpress [31] and collected by the authors of this study, as the original XML dataset used in ProbDD paper is not publicly available.

B. Evaluation Metrics

We measure the following aspects as metrics.

Final Size. This metric assesses the effectiveness of reduction. When reducing a list L with a certain property ψ, a smaller final list is preferred, indicating that more irrelevant elements have been successfully eliminated. In all benchmark suites, the metric is measured by *the number of tokens*.

Execution Time. The execution time of a minimization algorithm reflects its efficiency. A minimization algorithm taking less time is more desirable, and execution time is measured in *seconds*.

Query Number. This metric further evaluates the algorithm's efficiency. During the reduction process, each time a variant is produced, the algorithm verifies whether this variant still preserves the property ψ, referred to as a query. Since queries consume time, a lower query number is favorable.

P-value. We calculate the p-value via Wilcoxon signed-rank test [32] between every two algorithms, to investigate whether the performance differences are significant. In general, a p-value below 0.05 denotes a significant distinction between the two groups of data. Otherwise, the observed difference lacks statistical significance.

C. The Wrapping Frameworks

The `ddmin` algorithm and its variants usually serve as the fundamental algorithm. To apply them to a concrete scenario, an outer wrapping framework is generally needed to handle the structure of the input. In our evaluation, we choose the same wrapping frameworks as those used by ProbDD paper. For those tree-structured bug-triggering inputs, *i.e.*, BM_C and BM_{XML}, we use Picireny 21.8 [33], an implementation of HDD [13]. Picireny parses such inputs into trees, and then invokes Picire 21.8 [25], an open-sourced Delta Debugging library with `ddmin`, ProbDD and CDD implemented, to reduce each level of the trees. For software debloating on BM_{DBT}, Chisel [15] is employed, in which `ddmin`, ProbDD and CDD are integrated.

All experiments are conducted on a server running Ubuntu 22.04.3 LTS, with 4 TB RAM and two Intel Xeon Gold 6348 CPUs @ 2.60GHz. To ensure the reproducibility, we employ docker images to release the source code and the configuration. Each benchmark is reduced using a single thread. Following the ProbDD paper, we run each algorithm on each benchmark 5 times and calculate the geometric average results.

D. Reproduction Study of ProbDD

To comprehensively reproduce the results of ProbDD [16], we evaluate `ddmin` and ProbDD using three benchmark suites, containing a total of 76 benchmarks. Following the settings of ProbDD [16], we set the empirically estimated remaining rate as the initialization probability p_0, specifically, 0.1 for BM_C and BM_{DBT}, and 2.5e-3 for BM_{XML}. The detailed results are shown in Table IV and Table V.

Efficiency and Effectiveness. Through our reproduction study, we find that the performance of ProbDD aligns with the results reported in the original paper, showing that ProbDD is significantly more efficient than `ddmin`. Across three benchmark suites, ProbDD requires 27.01% less time and 52.44% fewer queries, with p-value being 3e-09 and 9e-14, respectively. Moreover, we assess the effectiveness by measuring the sizes of the final minimized results. The effectiveness of `ddmin` and ProbDD varies across each benchmark, but neither algorithm consistently outperforms the other, as substantiated by a p-value of 0.32, which is much higher than 0.05.

E. Impact of Randomness in ProbDD

Finding 3: Randomness has no significant impact on the performance of ProbDD.

In ProbDD, elements with different probabilities are sorted accordingly, while elements with the same probability are randomly shuffled. However, randomness alone intuitively does not ensure a higher probability of escaping local optima and the effect of this randomness on performance has not been thoroughly investigated.

To this end, we conduct an ablation study by removing such randomness, creating a variant called ProbDD-no-random. We evaluate this variant across all benchmarks. The results indicate that the randomness does not significantly impact performance. Specifically, in terms of final size, execution time, and query number, ProbDD-no-random achieves 236, 2,069, and 1,238 compared to 235, 2,189, and 1,309 of ProbDD, respectively. The p-values of 0.87, 0.15, and 0.10 indicate that the differences are not significant.

F. Bottleneck Overcome by ProbDD

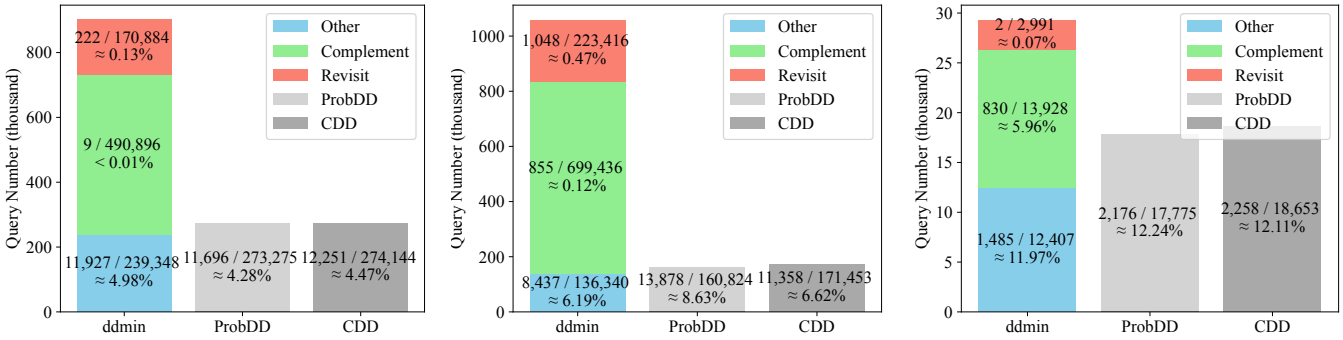
Finding 4: On tree-structured inputs, inefficient deletion attempts on complements and repeated attempts account for the bottlenecks of `ddmin`, which are overcome by ProbDD.

In the study of ProbDD, the authors demonstrate that ProbDD is more efficient than the baseline approach (`ddmin`) in tree-based reduction scenarios, where the inputs are parsed into tree representations before reduction. Therefore, to uncover the root cause of this superiority, we follow the same application scenario and analyze the behavior of ProbDD in reducing the tree-structured inputs.

To further understand why ProbDD is more efficient than `ddmin`, we conduct in-depth statistical analysis on the query number (number of deletion attempts). Intuitively, performance bottlenecks lie in those queries with low success rates, impairing `ddmin`'s efficiency. Existing studies [17], [18] also demonstrate the presence of queries with low success rates. Therefore, to qualitatively and quantitatively identify the exact bottlenecks impairing `ddmin`, we statistically analyze all the queries in `ddmin` and categorize them into three types:

TABLE IV: The final size, execution time and query number of `ddmin`, ProbDD and CDD on BM_C and BM_{DBT}. To address significant variations across benchmarks, the geometric mean rather than the arithmetic mean is employed, providing a smoother measure of the average.

	Benchmark	Original size (#)	Final size (#)			Execution time (s)			Query number		
			ddmin	ProbDD	CDD	ddmin	ProbDD	CDD	ddmin	ProbDD	CDD
BM_C	LLVM-22382	9,987	350	353	350	1,917	1,163	1,005	11,388	5,973	5,262
	LLVM-22704	184,444	786	764	745	27,924	12,418	11,371	52,412	15,425	14,025
	LLVM-23309	33,310	1,316	1,338	1,265	17,619	9,991	10,828	55,968	19,195	17,953
	LLVM-23353	30,196	321	336	324	3,117	1,874	1,400	11,719	5,757	4,492
	LLVM-25900	78,960	941	932	937	7,258	3,683	3,104	35,740	12,553	12,817
	LLVM-26760	209,577	520	503	498	13,123	5,876	5,210	30,063	9,261	9,792
	LLVM-27137	174,538	972	1,040	966	63,971	22,208	23,154	122,516	22,292	20,460
	LLVM-27747	173,840	431	463	510	6,545	4,238	2,932	20,000	8,193	5,992
	LLVM-31259	48,799	1,033	965	1,035	13,815	7,497	8,205	35,135	10,776	13,445
	GCC-59903	57,581	1,185	845	743	9,067	4,879	3,587	47,698	15,725	13,844
	GCC-60116	75,224	1,615	1,628	1,617	44,287	27,202	27,195	80,059	27,268	23,204
	GCC-61383	32,449	959	966	974	12,579	6,514	6,566	43,716	13,593	14,149
	GCC-61917	85,359	882	908	884	6,740	3,591	2,953	31,414	12,908	14,194
	GCC-64990	148,931	744	876	681	21,633	11,890	11,119	44,521	16,074	12,112
	GCC-65383	43,942	706	701	709	5,132	3,543	3,358	25,051	8,591	9,686
	GCC-66186	47,481	1,012	981	1,001	14,280	7,236	12,478	47,253	12,741	19,094
	GCC-66375	65,488	1,128	1,141	1,204	23,576	14,182	23,229	47,339	15,690	16,469
	GCC-70127	154,816	934	973	930	36,390	23,925	24,143	54,925	15,388	15,219
	GCC-70586	212,259	1,583	1,561	1,572	28,859	13,519	15,818	102,603	24,716	30,715
	GCC-71626	4,397	184	184	184	119	114	99	1,608	1,156	1,220
	Mean	64,599	777	775	760	10,486	5,828	5,676	34,013	11,652	11,512
BM_{DBT}	bzip2-1.0.5	70,530	20,710	20,747	20,756	137,463	105,782	92,058	54,034	23,240	19,846
	chown-8.2	43,869	9,087	9,303	9,310	38,902	25,616	7,625	50,487	8,278	8,208
	date-8.21	53,442	20,604	20,738	21,120	115,292	16,486	15,378	139,934	14,235	13,928
	grep-2.19	127,681	28,723	28,627	28,990	97,821	85,694	93,552	277,130	42,246	37,607
	gzip-1.2.4	45,929	17,065	17,068	17,077	73,403	55,520	75,913	147,035	27,569	61,032
	mkdir-5.2.1	34,801	8,625	8,782	8,418	3,227	2,428	1,877	11,969	2,836	2,099
	rm-8.4	44,459	8,507	8,467	8,461	12,087	5,008	5,109	33,171	5,097	5,057
	sort-8.16	88,068	14,893	14,843	15,834	60,631	61,739	21,948	119,150	18,711	7,914
	tar-1.14	163,296	20,411	20,713	20,592	115,234	95,765	77,910	200,394	14,384	12,095
	uniq-8.16	63,861	14,350	14,262	14,354	21,672	23,177	19,124	25,886	4,228	3,669
	Mean	65,151	15,080	15,152	15,235	43,827	28,505	21,782	72,140	11,803	10,686



(a) On BM_C (b) On BM_{DBT} (c) On BM_{XML}

Fig. 2: Visualization of queries within `ddmin`, ProbDD and CDD. In `ddmin`, three types of queries are displayed via stacked bars, the height of which denotes the query number. Within each bar, the number of successful queries, total queries and the corresponding success rate are annotated.

1) *Complement*: Queries attempting to remove the complement of a subset. According to `ddmin` algorithm, given a subset (smaller than half of the list L), it attempts to remove either the subset or its complement. However, evidence [18] shows that keeping a small subset and removing its complement is not likely to succeed, especially on structured inputs like programs.
2) *Revisit*: Queries attempting to remove the previously tried subset. After removing a subset, `ddmin` restarts the process from the first subset, leading to repeated deletion attempts on earlier subsets. Although the removal of one subset may allow another subset to be removable, such repetitions rarely succeed and thus offer limited improvement for the reduction [17].
3) *Other*: All other queries.

In addition to categorizing queries in `ddmin` into the above types, we also calculate the success rate of each type, aiming to reveal the bottlenecks of `ddmin`. Fig. 2 illustrates the distribution of queries for all types within `ddmin`, as well as the query number for ProbDD across all three benchmark suites.

On all benchmark suites, the number of successful queries in `ddmin` and ProbDD is remarkably similar, especially when

TABLE V: The final size, execution time and query number of ddmin, ProbDD and CDD on BM_{XML}. The last row shows the overall average across all three benchmark suites.

	Benchmark	Original size (#)	Final size (#)			Execution time (s)			Query number		
			ddmin	ProbDD	CDD	ddmin	ProbDD	CDD	ddmin	ProbDD	CDD
BM_{XML}	xml-071d221-1	20,090	10	15	20	73	114	144	29	50	69
	xml-071d221-2	20,387	13	14	20	155	146	243	60	69	117
	xml-1e9bc83-1	20,327	38	49	24	491	522	284	235	236	115
	xml-1e9bc83-2	20,222	79	78	76	1,391	814	867	725	390	384
	xml-1e9bc83-3	20,219	69	70	72	1,313	893	889	619	416	404
	xml-1e9bc83-4	19,985	156	139	143	3,911	1,939	2,521	1,943	935	1,173
	xml-1e9bc83-5	20,579	81	73	75	1,355	929	882	746	485	428
	xml-1e9bc83-6	19,880	127	126	124	3,563	1,852	1,548	1,907	964	749
	xml-1e9bc83-7	20,297	111	114	111	3,419	1,684	2,330	1,757	827	931
	xml-1e9bc83-8	20,327	100	107	100	2,862	1,636	1,446	1,451	751	592
	xml-1e9bc83-9	20,330	128	73	128	2,850	1,230	2,494	1,437	561	832
	xml-2d4ec80-1	20,129	76	72	76	570	522	791	384	304	354
	xml-327c8af-1	20,207	55	55	55	864	625	958	527	319	392
	xml-3398ac2-1	20,414	45	44	48	345	383	605	225	192	268
	xml-3398ac2-2	19,290	48	48	48	613	492	578	358	255	238
	xml-3398ac2-3	20,222	62	62	62	632	574	764	347	276	306
	xml-3398ac2-4	19,913	111	112	111	1,419	1,197	1,722	802	584	680
	xml-3398ac2-5	20,477	44	38	44	850	531	633	507	269	261
	xml-4c99b96-1	20,513	80	78	80	1,427	1,126	1,385	699	452	438
	xml-4c99b96-10	20,522	46	47	46	1,012	760	873	443	299	256
	xml-4c99b96-11	19,901	53	54	53	868	661	791	357	252	236
	xml-4c99b96-12	19,775	102	104	102	2,429	1,592	2,422	1,077	626	773
	xml-4c99b96-13	20,114	60	61	60	1,132	870	989	494	335	309
	xml-4c99b96-14	19,970	102	103	102	2,147	1,687	1,534	987	600	504
	xml-4c99b96-15	20,138	46	46	46	843	788	729	381	297	244
	xml-4c99b96-16	20,126	67	68	70	1,366	1,189	1,125	658	414	409
	xml-4c99b96-17	20,210	67	68	70	1,360	1,310	1,120	658	414	409
	xml-4c99b96-18	20,390	28	28	25	724	492	410	324	164	137
	xml-4c99b96-19	20,192	81	82	81	1,729	1,798	1,895	793	540	710
	xml-4c99b96-2	20,750	53	54	53	878	1,017	912	362	281	318
	xml-4c99b96-3	20,015	67	68	67	1,102	1,254	900	483	357	337
	xml-4c99b96-4	20,201	67	67	67	1,097	1,238	860	482	365	328
	xml-4c99b96-5	20,279	63	62	60	1,170	1,155	1,029	515	347	364
	xml-4c99b96-6	19,973	81	82	87	1,258	1,338	1,056	511	381	441
	xml-4c99b96-7	19,973	115	115	115	3,399	2,560	1,535	1,485	811	676
	xml-4c99b96-8	20,579	42	41	42	926	810	687	422	254	323
	xml-4c99b96-9	20,075	53	53	53	884	940	650	368	278	265
	xml-8ede045-1	20,192	48	61	69	1,152	1,232	1,137	499	389	487
	xml-8ede045-2	20,177	17	22	34	316	335	360	132	107	164
	xml-8ede045-3	20,393	31	31	31	476	431	313	199	134	134
	xml-8ede045-4	20,123	31	25	31	578	472	435	250	151	181
	xml-8ede045-5	20,051	17	20	17	185	246	204	61	72	80
	xml-8ede045-6	20,636	73	80	76	1,365	1,479	1,545	552	469	700
	xml-8ede045-7	20,054	106	106	106	2,851	1,686	1,237	1,447	614	634
	xml-8ede045-8	20,177	76	78	76	1,397	1,327	1,060	597	449	475
	xml-f053486-1	20,030	10	10	10	101	134	76	31	41	28
	Mean	20,190	56	56	58	972	819	821	453	314	327
All	Mean	31,989	233	235	237	2,999	2,189	2,102	2,752	1,309	1,320

contrasted with the substantial difference in the total number of queries. Specifically, on BM_C, ddmin achieves $222 + 9 + 11,927 = 12,158$ successful queries, closely matching the 11,696 successful queries from ProbDD. Similarly, on BM_{DBT} and BM_{XML}, ddmin performs $1,048 + 855 + 8,437 = 10,340$ and $2 + 830 + 1,485 = 2,317$ successful queries, respectively, both closely aligning with the 13,878 and 2,176 successful queries achieved by ProbDD. Besides, ddmin always performs significantly more failed queries, resulting in a larger total query number and thus a longer execution time, as previously discussed in § IV-D.

On all benchmark suites, a large portion of ddmin's queries is categorized as *Complement* and *Revisit*; however, they both have a notably low success rate. For instance, on BM_C, out of a total of 901,128 queries, *Complement* and *Revisit* account for 490,896 (54.48%) and 170,884 (18.96%), respectively. Within such queries in *Complement* and *Revisit*, merely 9 (<0.01%) and 222 (0.13%) queries succeed, *i.e.*, only a tiny portion of attempts successfully reduce elements. These success rates are far less than those of queries within *Other* (4.98%), as well as those of ProbDD (4.28%). On the other benchmark suites, a similar phenomenon is observed.

Queries within *Complement* and *Revisit* categories constitute a large portion yet prove to be largely inefficient, wasting a significant amount of time and resources. On the contrary, those in *Other* achieve a much higher success rate, on par with that of ProbDD, and are responsible for most of the successful deletions. Therefore, we believe that these two categories, where queries are inefficient, are the main bottlenecks behind ddmin's low efficiency. However, these bottlenecks are absent in ProbDD, as it does not consider complements of subsets and previously tried subsets for deletion.

G. 1-Minimality of ProbDD?

Finding 5: Improving efficiency by avoiding ineffective attempts presents a trade-off by not ensuring 1-minimality, while such limitation can be mitigated by iteratively running the reduction algorithm until a fixpoint is reached.

Although ProbDD avoids *Revisit* queries to enhance efficiency, some reduction potentials may be missed, as the deletion of a certain subset may enable a previously tried subset to become removable. Therefore, a limitation of ProbDD lies in that it increases efficiency by sacrificing 1-minimality. To substantiate this limitation, we examine how frequently ProbDD generates a list that is not 1-minimal, *i.e.*, can be further reduced by removing a single element. For instance, statistical analysis on BM_C reveals that among 6,871 invocations of ProbDD, 76 of them fail to generate a 1-minimal result, accounting for 1.1%. For these failed invocations, an average of 1.49 elements (tree nodes) can be further removed via single-element deletion.

However, such limitation is not apparent across all benchmark suites, as the results from ProbDD are not consistently larger than those from `ddmin`. Our further investigation reveals that these benchmarks are reduced on wrapper frameworks Picireny and Chisel. Both frameworks employ iterative loops to achieve a fixpoint, effectively reducing some elements missed in the first iteration.

V. Implications: A Counter-Based Model

Building on the aforementioned demystification of ProbDD, we discover that probability can be optimized away, and subset size can be pre-computed. Hence, we propose Counter-Based Delta Debugging (CDD), to reduce the complexity of both the theory and implementation of ProbDD, and validate the correctness of our prior theoretical proofs.

Subset size pre-calculation. Based on Lemma III.3 in § III-B, the size for each round can be pre-calculated. Therefore, as shown at line 12 – line 15 in Algorithm 2, we utilize the current round r and the initial probability p_0 to determine the subset size s. The size of the selected subset decreases as the round counter increases. This is intuitively reasonable since, after a sufficient number of attempts on a large size have been made, it becomes more advantageous to gradually reduce the subset size for future trials. Furthermore, this trend aligns well with that of ProbDD, in which probabilities of elements gradually increase, resulting in a smaller subset size.

Main workflow. The simplified ProbDD is illustrated in Algorithm 2, from line 1 to line 11. Before each round, the CDD pre-calculates the subset size on line 3 and then partitions L using this size on line 4. Then, similar to `ddmin`, it attempts to remove each subset on line 5 – line 8. The subset size continuously decreases until it reaches 1, meaning that each element will be individually removed once.

Revisiting the running example. Returning to Table III, under the same conditions, CDD achieves the same results as ProbDD but without the need for probability calculations.

Algorithm 2: `CDD` (L, ψ)

```
Input: L: a list of element to be reduced.
Input: ψ : 𝕃 → 𝔹: the property to be preserved by L.
Input: p_0: the initial probability given by the user.
Output: the minimized list that still exhibits the property ψ .
1  r ← 0 // The round number, initially 0.
2  do
       // Compute subset size by round number
3      s ← ComputeSize (r, p_0)
       /* Partition L into subsets with s elements. If it does
          not divide evenly, leave a smaller remainder as the
          final subset.                                        */
4      subsets ← Partition (L, s)
5      foreach subset ∈ subsets do
6          temp ← L \ subset
           // Remove subset if it is removable
7          if ψ(temp) is true then
8              L ← temp
       // Update the r and move to next round.
9      r ← r + 1
10 while s > 1
11 return L
12 Function ComputeSize(r, p_0):
       Input: r: the current round number.
       Input: p_0: the initial probability given by the user.
       Output: The size of the subset to be used in the current
               round.
       // Calculate the estimated probability of round r
13     p_r ← p_0 × 1.582^r
       // Calculate corresponding subset size of round r
14     s_r = arg max_{s∈ℕ+} s × (1 − p_r)^s
15     return s_r
```

This is because both the probability and subset size s can be directly determined from the round number r.

Evaluation. As shown in Table IV and Table V, CDD outperforms `ddmin` *w.r.t.* efficiency, with 29.91% less time and 52.04% fewer queries. Meanwhile, CDD performs comparably to ProbDD *w.r.t.* final size, execution time and query number, with a p-value of 0.42, 0.29 and 0.70, respectively, indicating insignificance between these two algorithms. CDD is expected to perform on par with ProbDD since it is designed to provide further insight and simplify the intricate design of ProbDD, rather than to surpass its capabilities. Furthermore, its comparable performance to ProbDD further validates the non-necessity of randomness and our assumption in Lemma III.1.

Bottleneck and 1-minimality. Revisiting the bottlenecks presented in Fig. 2, CDD possesses a query number and success rate close to those of ProbDD, indicating that CDD also overcomes the bottlenecks of `ddmin`. Additionally, similar to ProbDD, 1-minimality is absent in CDD, although iterations help mitigate this issue.

Finding 6: CDD always achieves comparable performance to ProbDD, which further supports our previous findings, including the theoretical simplifications regarding size and probability, analysis of randomness, bottlenecks, and 1-minimality.

VI. Limitations and threats to validity

In this section, we discuss the limitations of CDD, and potential factors that may impair the validity of our experimental results.

A. Limitations

As discussed in § IV-G, compared to `ddmin`, neither ProbDD nor CDD guarantees 1-minimality. This limitation arises because after successfully removing a subset, `ddmin` restarts the process from the first subset, whereas ProbDD and CDD continue from the next subset, skipping all the previously tried subsets. Therefore, although ProbDD and CDD complete the reduction process more quickly, they may miss certain reduction opportunities and produce larger results than `ddmin`.

However, reduction and debloating tools generally invoke these reduction algorithms in iterative loops until a fix-point is reached, gradually refining the results and mitigating limitations, as mentioned in § IV-G. Table IV and Table V further support this point by showing that with multiple iterations, ProbDD and CDD achieve significantly higher efficiency compared to `ddmin`, while still producing results that are comparable to `ddmin` w.r.t. effectiveness.

B. Threats to validity

For internal validity, the main threat comes from the potential impact from the assumption, as discussed in § III-A1. Specifically, without assuming that the number of elements in L is always divisible by the current subset size, we could not further refine the mathematical model of ProbDD to achieve a simpler representation. However, such assumption might impact the actual performance, potentially negating the benefits of our simplification. To this end, we conduct extensive empirical experiments, demonstrating that CDD, the simplified algorithm derived from this assumption, exhibits no significant difference from ProbDD.

For external validity, the threat lies in the generalizability of our findings across application scenarios. To mitigate this threat, we perform experiments on 76 benchmarks, including C programs triggering real-world compiler bugs, XML inputs crashing XML processing tools, and benchmarks from software debloating tasks. These benchmarks have covered various use scenarios of minimization algorithms.

VII. Related Work

In this section, we discuss related work of test input minimization around three aspects: effectiveness, efficiency, and the utilization of domain knowledge.

Effectiveness. Test input minimization is an NP-complete problem, in which achieving the global minimum is usually infeasible. Therefore, existing approaches to improving effectiveness mainly aim to escape local minima by performing more exhaustive searches. Since enumerating all possible subsets is infeasible, Vulcan [34] and C-Reduce [14] enumerate all combinations of elements within a small sliding window, and exhaustively attempt to delete each combination, resulting in smaller final program sizes. In contrast, ProbDD and CDD do not exhibit clear actions targeted at breaking through local optima, suggesting they cannot achieve better effectiveness than `ddmin`, as aligned with our evaluation in § IV.

Efficiency. If parallelism is not considered, the core of boosting efficiency is the enhanced capability to avoid relatively inefficient queries. For example, Hodovan and Kiss [18] proposed disregarding attempts to remove the complement of subsets, the success rate of which is unacceptably low in some scenarios. Besides, Gharachorlu and Sumner [17] proposed One Pass Delta Debugging (OPDD), which continues with the subset next to the deleted one, rather than starting over from the first subset. This optimization also avoids some redundant queries in `ddmin`, reducing runtime by 65%. As revealed by our analysis, these two above-mentioned optimizations are implicitly incorporated within ProbDD and CDD, and thereby contributing to their higher efficiency than `ddmin`.

Utilization of domain knowledge. There is an inherent trade-off between effectiveness and efficiency in test input minimization. For the same algorithm, achieving a better result, *i.e.*, a smaller local optimum, requires more queries to be spent on trial and error. However, employing domain knowledge [14], [35]–[37] can still improve the overall performance. For instance, J-Reduce is both more effective and efficient than HDD in reducing Java programs, as it escapes more local optima by program transformations while simultaneously avoiding more inefficient queries via semantic constraints, leveraging the semantics of Java. Our analysis on ProbDD indicates that the probabilities primarily function as counters and do not utilize or effectively learn the domain knowledge of an input. Besides, the evaluation on CDD, a simplified algorithm without utilizing probability, demonstrates that prioritizing elements via such probabilities does not yield significant benefits, thus validating our analysis.

VIII. Conclusion

This paper conducts the first in-depth analysis of ProbDD, which is the state-of-the-art variant of `ddmin`, to further comprehend and demystify its superior performance. With theoretical analysis of the probabilistic model in ProbDD, we reveal that probabilities essentially serve as monotonically increasing counters, and propose CDD for simplification. Evaluations on 76 benchmarks from test input minimization and software debloating confirm that CDD performs on par with ProbDD, substantiating our theoretical analysis. Furthermore, our examination on query success rate and randomness uncovers that ProbDD's superiority stems from skipping inefficient queries. Finally, we discuss trade-offs in `ddmin` and ProbDD, providing insights for future research and applications of test input minimization algorithms.

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] GCC. (2020) A guide to testcase reduction. Accessed: 2023-04-30. [Online]. Available: https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[3] LLVM. (2022) How to submit an llvm bug report. Accessed: 2023-04-30. [Online]. Available: https://llvm.org/docs/HowToSubmitABug.html

[4] WebKit. (2001) Webkit: Test case reduction. Accessed: 2023-04-30. [Online]. Available: https://webkit.org/test-case-reduction/

[5] ASF Bugzilla. (2001) ASF bugzilla: Bug writing guidelines. Accessed: 2023-04-30. [Online]. Available: https://bz.apache.org/bugzilla/page.cgi?id=bug-writing.html

[6] Bugzilla. (2001) Bugzilla: Reporting a new bug. Accessed: 2023-04-30. [Online]. Available: https://bugzilla.readthedocs.io/en/5.2/using/filing.html#reporting-a-new-bug

[7] A. Donaldson and D. MacIver. (2021, May) Test Case Reduction: Beyond Bugs. [Online]. Available: https://blog.sigplan.org/2021/05/25/test-case-reduction-beyond-bugs

[8] T. L. Wang, Y. Tian, Y. Dong, Z. Xu, and C. Sun, "Compilation consistency modulo debug information," in *ASPLOS '23: 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Vancouver, 25 March 2023 - 29 March 2023*. ACM, 2023.

[9] Y. Tian, Z. Xu, Y. Dong, C. Sun, and S. Cheung, "Revisiting the evaluation of deep learning-based compiler testing," in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 2023, pp. 4873–4882. [Online]. Available: https://doi.org/10.24963/ijcai.2023/542

[10] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 849–863.

[11] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpiński, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1017–1032.

[12] 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.

[13] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 142–151.

[14] 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.

[15] 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.

[16] 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.

[17] G. Gharachorlu and N. Sumner, "Avoiding the familiar to speed up test case reduction," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 426–437.

[18] R. Hodován and Á. Kiss, "Practical improvements to the minimizing delta debugging algorithm." in *ICSOFT-EA*, 2016, pp. 241–248.

[19] X. Zhou, Z. Xu, M. Zhang, Y. Tian, and C. Sun, "Wdd: Weighted delta debugging," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, 2025.

[20] G. Wang. (2021) Probdd. Accessed: 2023-04-30. [Online]. Available: https://github.com/Amocy-Wang/ProbDD

[21] M. Zhang, Z. Xu, Y. Tian, X. Cheng, and C. Sun, "Artifact for "toward a better understanding of probabilistic delta debugging"," 2024. [Online]. Available: https://zenodo.org/records/14425530

[22] A. Christi, A. Groce, and R. Gopinath, "Resource adaptation via test-based software minimization," in *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2017, pp. 61–70.

[23] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 243–252.

[24] ——, "Cause reduction: delta debugging, even without bugs," *Software Testing, Verification and Reliability*, vol. 26, no. 1, pp. 40–68, 2016.

[25] A. Kiss, R. Hodován, and D. Vince. (2016) Picire. Accessed: 2023-04-30. [Online]. Available: https://github.com/renatahodovan/picire

[26] M. Pelikan, D. E. Goldberg, E. Cantú-Paz *et al.*, "Boa: The bayesian optimization algorithm," in *Proceedings of the genetic and evolutionary computation conference GECCO-99*, vol. 1. Citeseer, 1999, pp. 525–532.

[27] Z. Xu, Y. Tian, M. Zhang, G. Zhao, Y. Jiang, and C. Sun, "Pushing the limit of 1-minimality of language-agnostic program reduction," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. [Online]. Available: https://doi.org/10.1145/3586049

[28] 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 Trans. Softw. Eng. Methodol.*, Aug. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3690631

[29] C. Qian, H. Hu, M. Alharthi, S. P. H. Chung, T. Kim, and W. Lee, "Razor: A framework for post-deployment software debloating." in *USENIX Security Symposium*, 2019, pp. 1733–1750.

[30] M. Alhanahnah, R. Jain, V. Rastogi, S. Jha, and T. Reps, "Lightweight, multi-stage, compiler-assisted application specialization," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 251–269.

[31] S. Li and M. Rigger, "Finding xpath bugs in xml document processors via differential testing," *arXiv preprint arXiv:2401.05112*, 2024.

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

[33] A. Kiss, R. Hodován, and D. Vince. (2016) Picireny. Accessed: 2023-04-30. [Online]. Available: https://github.com/renatahodovan/picireny

[34] 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.

[35] 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.

[36] ——, "Logical bytecode reduction," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1003–1016.

[37] 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*. New York, NY, USA: ACM, 2024, p. 13.