Dr. DD: 1-Minimal Isolation of Failure Causes via Deferred Restarts

Aarush Kumbhakern
Ashoka University, India
aarush.kumbhakern_ug25@ashoka.edu.in

Feiyang Chen
University of Sydney, Australia
fche0060@uni.sydney.edu.au

Danushka Liyanage
University of Sydney, Australia
danushka.liyanage@sydney.edu.au

Xi Wu
University of Sydney, Australia
xi.wu@sydney.edu.au

Mohammad Amin Alipour
University of Houston, USA
maalipou@central.uh.edu

Rahul Gopinath
University of Sydney, Australia
rahul.gopinath@sydney.edu.au

***Abstract*—Delta debugging is a widely used algorithm to reduce failure-inducing inputs in software testing. It guarantees 1-minimality but suffers from quadratic worst-case complexity due to repeated restarts at every partition level, limiting its scalability.**

Re-examining `ddmin`, we show that restarts are only required at the single-element level to preserve 1-minimality. Restarts are redundant at coarser granularities and can be deferred without affecting the 1-minimality guarantee. We further show that the quadratic worst case arises from causal chains, not from restarts.

These insights lead to `drdd`, a drop-in replacement for `ddmin` that preserves 1-minimality while avoiding unnecessary restarts. `drdd` also exposes a restart budget R, allowing a trade-off between minimality and linear worst-case performance.

We evaluate `drdd` on `ffmpeg`, XML, `binutils`, and `crashjs` inputs. On XML inputs, `drdd` reduces oracle calls to 1.7% of `ddmin`^{Y} and achieves a $6.0\times$ speedup, and shows the largest wall-clock speedup on `crashjs`. `drdd` performs well when reduction is possible and shows smaller gains on inputs with limited or no reducibility. Across all subjects, `drdd` matches the reduction quality of `ddmin`^{Y} and avoids the degradation observed in ProbDD and CDD.

***Index Terms*—Delta debugging, input minimization**

I. Introduction

Input reduction is a fundamental technique in software engineering. Reduced inputs help 1) reduce execution time of test cases [1], 2) de-duplicate bugs found during fuzzing [2], 3) improve test case comprehension by removing unimportant parts of the input [3], 4) improve fault localization [4], and 5) improve constraint solving [5].

Delta debugging, introduced by Zeller and Hildebrandt [6], is the de facto technique for structure-agnostic input reduction. Its classical algorithm, `ddmin`, systematically removes subsets of the input until it converges on a 1-minimal failure-inducing input. However, `ddmin` suffers from a well-known $O(n^2)$ worst case, which severely limits its applicability to large inputs such as those produced by coverage-guided fuzzers.

This paper shows that the inefficiency of `ddmin` is not solely due to its worst-case bound, but to how restarts are applied throughout the reduction process. To explain why, we need two key terms: a *restart* resets the partition size to the start of the current scan phase after a successful reduction; an output is *1-minimal* if no single element can be removed while still preserving the failure. While the quadratic worst case is unavoidable in the presence of causal chains, `ddmin` performs restarts at every partition level. We show that restarts are only necessary at the single-element level; at coarser granularities they introduce redundant work without contributing to the 1-minimality guarantee. By deferring restarts to the single-element level, we significantly reduce the cost of restarts on practical inputs, where causal chains are limited.

We arrive at this result through a careful re-examination of `ddmin` and its variants. Our analysis surfaces several properties that are implicit in prior work but not explicitly articulated, and uses them to derive Deferred Restart Delta Debugging (`drdd`), a simple and efficient variant of `ddmin`. `drdd` preserves 1-minimality without relying on prior assumptions about input reducibility, and reduces unnecessary oracle calls when meaningful reduction is possible.

Explaining why subset test is unnecessary for 1-minimality. The original `ddmin` [6] operates in two phases: it first tests each partition *subset* to see whether the subset alone reproduces the failure, then tests each *complement* (the input with one partition removed). The subset test can accelerate reduction when the failure-inducing region is small and contiguous, but it is not required for the 1-minimality guarantee. Later published versions of this algorithm [7], [8] omit the subset test entirely, without explaining why the omission preserves the 1-minimality guarantee or what is lost by omitting it. We make this trade-off explicit: omitting the subset test improves efficiency when inputs are hard to reduce, but can miss reductions on inputs where a contiguous subset alone triggers the failure. Since structure-agnostic reducers cannot guarantee partition alignment with such regions, the subset test rarely pays off in practice.

Restarts are strictly necessary only at the last level. When `ddmin` makes progress at some partition level, it restarts that level from the beginning. The motivation is to handle *causal chains*: if removing element B makes element A removable, restarting at the same level allows A to be removed without descending to a finer granularity. However, if no restart at a given level can succeed (because no partition boundary aligns with a removable element), these restarts are wasted. We show

that restarts are only strictly necessary at the single-element level to preserve 1-minimality, since causal chains are resolved only at that granularity. At coarser partition sizes, restarts do not contribute to the 1-minimality guarantee, though they may affect which 1-minimal fixed point is reached. Deferring restarts to the single-element level avoids this overhead.

The $O(n^2)$ worst case is structurally a causal chain. Zeller [6] proves the $O(n^2)$ worst-case bound (Prop. 12): This occurs at single-element granularity, when only the last complement always fails, forcing the algorithm to remove one element at a time. We observe that this pattern is structurally identical to a causal chain: element i can only be removed after element $i+1$ has already been removed. This interpretation, while implicit in the proof, has not been made explicit.

Exposing the restart budget as a parameter. Bounding single-element scans by a parameter R yields a family interpolating between full 1-minimality (quadratic worst case) and linear performance without the guarantee. We instantiate this as `drdd`, which defers restarts to the single-element level. In contrast, ProbDD [9] and CDD [10] eliminate restarts entirely, sacrificing the 1-minimality guarantee.

We compare `ddmin`^{Y}, `drdd`, ProbDD, and CDD on binary and structured inputs from `ffmpeg`, XML, `binutils`, and `crashjs`, evaluating all algorithms independently to isolate algorithmic behavior. `drdd` matches `ddmin`^{Y} in reduction quality while using substantially fewer oracle calls across all subjects. On `ffmpeg` and XML, `drdd` matches or outperforms ProbDD and CDD in oracle calls while producing smaller outputs and preserving 1-minimality; on `binutils` and `crashjs`, ProbDD and CDD use fewer oracle calls but produce non-1-minimal outputs.

Contributions:

- **Revisiting delta debugging.** We clarify the roles of the subset test, restart mechanism, and causal chains, making explicit properties implicit in prior analyses.
- **`drdd`.** A drop-in replacement for `ddmin` that defers restarts to the single-element level, preserving 1-minimality while reducing unnecessary oracle calls.
- **A unified design space.** We show that `ddmin`, `ddmin`^{Y}, `drdd`, ProbDD, and CDD can be understood within a common framework.
- **Empirical evaluation.** `drdd` matches `ddmin`^{Y} in reduction quality while achieving strong efficiency gains when reduction is possible.

II. Delta Debugging with ddmin

Let $I = \langle \Delta_1, \Delta_2, \ldots, \Delta_n \rangle$ be an input sequence and $test : \mathcal{I} \to \{\checkmark, \mathsf{X}, \mathbf{?}\}$ a deterministic oracle with $test(I) = \checkmark$ and $test(\emptyset) = \mathsf{X}$. $\checkmark$ means the failure is preserved; X means it is not. The algorithm can be applied to any program for which such a deterministic oracle can be defined, treating the input as an opaque byte sequence. The goal is a reduced input $I' \subseteq I$ such that $test(I') = \checkmark$ and $\forall \Delta_i \in I',\ test(I' \setminus \{\Delta_i\}) = \mathsf{X}$ (*1-minimality*)[1].

[1] $\Psi(c_x) = ?$ is treated as equivalent to $\Psi(c_x) = \mathsf{X}$ by `ddmin`.

A. Classical `ddmin` Algorithm

Given Ψ and c_x with $\Psi(c_x) = \checkmark$, the goal is $c'_x = \texttt{ddmin}(c_x)$ that is 1-minimal. $\texttt{ddmin}(c_x) = \delta_2(c_x, 2)$, where $\delta_2(c'_x, n)$ denotes a reduction step on input c'_x starting with n partitions, where

$$\delta_2(c'_x, n) = \begin{cases} \delta_2(\Delta_i, 2) & \text{if } \exists i : test(\Delta_i) = \checkmark \\ \delta_2(\nabla_i, (n-1) \vee 2) & \text{elif } \exists i : \Psi(\nabla_i) = \checkmark \\ \delta_2(c'_x, |c'_x| \wedge 2n) & \text{elif } n < |c'_x| \\ c'_x & \text{else.} \end{cases}$$

in order, where $\nabla_i = c'_x \setminus \Delta_i$, $c'_x = \bigcup_{i=1}^{n} \Delta_i$, all Δ_i are pairwise disjoint, and $\forall \Delta_i : |\Delta_i| \approx |c'_x|/n$.

The subset phase is rarely useful in practice, so later versions omit it.

B. `ddmin`^{Y}: Simplified `ddmin` Algorithm

In *Why Programs Fail* [7][2] and later in Fuzzingbook [8], Zeller presents a simplified version of `ddmin`, which we call `ddmin`^{Y}, obtained from the declarative form above by dropping the first case entirely:

$$\delta_2(c'_x, n) = \begin{cases} \delta_2(\nabla_i, (n-1) \vee 2) & \text{if } \exists i : \Psi(\nabla_i) = \checkmark \\ \delta_2(c'_x, |c'_x| \wedge 2n) & \text{elif } n < |c'_x| \\ c'_x & \text{else.} \end{cases}$$

The algorithm scans through complements at decreasing partition sizes, restarting to $n = 2$ (i.e., $S = |I|$) whenever a complement is accepted. It terminates when no complement succeeds at any partition size, at which point the output is 1-minimal.

C. Key Properties

Classical `ddmin` has the following properties:

- **1-minimality:** `ddmin` guarantees that no single delta can be removed from the minimized input while preserving the failure (Proposition 11 of Zeller and Hildebrandt [6]).
- **Best-case complexity:** $O(\log |I|)$ oracle calls when there is a single failure-inducing delta and every test case containing it also fails.
- **Worst-case complexity:** $|I|^2 + 3|I|$ oracle calls (Proposition 12 of [6]). The quadratic cost arises when $n = |I|$ and only the last complement succeeds, forcing one-element reductions until no further removal is possible.

  Causal chains. We say Δ_i causally depends on Δ_j if $\Psi(I \setminus \{\Delta_i\}) = \mathsf{X}$ but $\Psi(I \setminus \{\Delta_i, \Delta_j\}) = \checkmark$. A sequence $\Delta_1, \ldots, \Delta_k$ forms a causal chain if each Δ_i depends on Δ_{i+1}. Such chains force one-element progress per scan, yielding $\sum_{i=1}^{n} i = O(n^2)$ oracle calls.

III. drdd Algorithm

Let us first consider a simplified rendering of `ddmin`.

[2] List 5.2

Algorithm 1 COMPLEMENT_SCAN

```
function COMPLEMENT_SCAN(I, S, test)
    prefix ← ∅
    for i ← 0 to |I| − 1 step S do
        stitched ← prefix ‖ I[i + S : |I|]
        if test(stitched) = accept then
            return stitched                          ▷ remove block i
        else
            prefix ← prefix ‖ I[i : i + S]           ▷ block i required
    return prefix                  ▷ no complement accepted; prefix = I
```

A. Iterative Form of $\texttt{ddmin}^Y$

Algorithm 1 scans through the input in blocks of size S, attempting to remove each block by testing the complement (the input with that block omitted). Algorithm 2 drives this scan, halving S after each pass and restarting to $S = |I|$ whenever a block is removed.

Algorithm 2 $\texttt{ddmin}^Y$

```
function DDMIN(I, test)
    S ← |I|
    while true do
        S ← ⌊S/2⌋                          ▷ compute partition size
        if S = 0 then
            break
        I′ ← COMPLEMENT_SCAN(I, S, test)
        if I′ ≠ I then
            S ← |I|                                  ▷ causal restart
        I ← I′
    return I
```

Algorithm 2 recasts $\texttt{ddmin}^Y$, making explicit the structural point where drdd diverges. A causal restart allows the algorithm to resolve causal chains: removing one block may enable removal of an earlier block. However, restarts are expensive and only strictly necessary at single-element granularity. Deferring all restarts to that level gives drdd.

B. drdd Overview

Algorithm 3 is a non-early-exit variant of complement_scan that tries every block before returning, accumulating all removable blocks in a single pass. Algorithm 5 uses this to run coarse complement passes without restarting, deferring restarts to the bounded single-element scan in Algorithm 4.

drdd exposes the restart budget as parameter R, and two additional parameters S^{max} and S^{min} controlling the partition-size range of the coarse sweep.

C. Key Properties

The drdd algorithm has the following properties.

Best case. The best-case complexity matches that of $\texttt{ddmin}^Y$. When the input contains a single failure-inducing region that can be isolated by complement binary search, the main loop isolates the single element in $O(\log_2 |I|)$ tests.

1-minimality by default. With $R = |I|$ (the default), drdd terminates only when a full *single-element scan* (a complement sweep at partition size $S = 1$, testing removal of each element individually) produces no further reduction, which is exactly

Algorithm 3 FULL_COMPLEMENT_SCAN

```
function FULL_COMPLEMENT_SCAN(I, S, test)
    prefix ← ∅
    for i ← 0 to |I| − 1 step S do
        stitched ← prefix ‖ I[i + S : |I|]
        if test(stitched) = accept then
            pass                   ▷ block i dropped; no return
        else
            prefix ← prefix ‖ I[i : i + S]           ▷ block i required
    return prefix                  ▷ return after full complement scan
```

Algorithm 4 CAUSAL_CHAIN_SCAN: Bounded fixed-point scan at single-element granularity

```
function CAUSAL_CHAIN_SCAN(I, test, R)
    count ← R
    while count ≠ 0 do
        I′ ← FULL_COMPLEMENT_SCAN(I, 1, test)
        if I′ = I then break
        I ← I′
        count ← count − 1
    return I
```

the definition of 1-minimality (see Proposition 1). Note that 1-minimality does not imply a unique output: multiple 1-minimal fixed points may exist, and greedy algorithms with different restart policies may converge to different ones. A coarse-grained restart can therefore affect which fixed point is reached, even though it is not required for the guarantee itself.

Proposition 1 (1-minimality)**.** *Let I_0 be any input with $test(I_0) = ✓$ and $test(\emptyset) = ✗$. Then $\texttt{drdd}(I_0, test, R{=}|I_0|)$ terminates and returns a 1-minimal input.*

Proof. Let $I^{(k)}$ denote the value of the current input at the start of the k-th iteration of CAUSAL_CHAIN_SCAN, with $I^{(1)} = I_0$. Since every call to FULL_COMPLEMENT_SCAN returns a subset of its input, the coarse phase only removes elements; hence $test(I^{(k)}) = ✓$ is maintained as an invariant for all k.

Termination. Each iteration calls FULL_COMPLEMENT_SCAN$(I^{(k)}, 1, test)$, producing $I^{(k+1)}$. If $I^{(k+1)} = I^{(k)}$, the loop breaks. Otherwise $|I^{(k+1)}| \le |I^{(k)}| - 1$, so $|I^{(k)}|$ strictly decreases by at least 1 per restarting iteration. Since $|I^{(k)}| \ge 0$, there are at most $|I_0|$ restarting iterations before the loop must break. With $R = |I_0|$, the budget is never exhausted before this occurs, so the loop always exits via the break.

1-minimality. Let I^* be the output when the loop breaks,

Algorithm 5 drdd: Delta Debugging

```
function DRDD(I, test, R=|I|, S^max=|I|, S^min=1)
    S ← |I|
    while true do
        S ← S//2                          ▷ replace: COMPUTE_SIZE
        if S ≤ S^min then                 ▷ Ignore small partition sizes
            break                                    ▷ Stop at this size
        if S ≤ S^max then
            I ← FULL_COMPLEMENT_SCAN(I, S, test)
    return CAUSAL_CHAIN_SCAN(I, test, R)
```

Algorithm 6 PROBDD: Probabilistic Delta Debugging

```
function PROBDD(I, test)
    I_c ← I
    p_i ← 0.1 for all Δ_i ∈ I_c                ▷ 0.1 is an example.
    while ∃ Δ_i ∈ I_c : p_i < 1 do
        I_d ← SAMPLE(I_c)
        I_r ← I_c \ I_d
        if test(I_r) = accept then
            I_c ← I_r
        else
            UPDATEPROBS(I_d)
    return I_c
```

Algorithm 7 SAMPLE: Select elements for removal

```
function SAMPLE(I_c)
    L ← {i : i ∈ I_c} sorted by ascending p_i
    S ← ∅
    E* ← 0
    for i ∈ L do
        S ← S ∪ {i}
        E ← |S| · ∏_{s∈S}(1 − p_s)             ▷ Expected elements removed
        if E < E* then
            S ← S \ {i}
            break
        E* ← E
    return S
```

Algorithm 8 UPDATEPROBS: Bayesian update after rejected deletion

```
function UPDATEPROBS(I_d)
    r ← (1 − ∏_{i∈I_d}(1 − p_i))^{−1}          ▷ Update ratio
    for i ∈ I_d : p_i < 1 do
        p_i ← r · p_i
    if |I_d| = 1 then
        p_i ← 1 for i ∈ I_d                    ▷ Singleton must be required
```

i.e., FULL_COMPLEMENT_SCAN$(I^*, 1, test)$ returns I^* unchanged. By Algorithm 3, a complement scan at $S = 1$ tests $I^* \setminus \{\Delta_i\}$ for *every* $\Delta_i \in I^*$. Since $test(I^*) = \checkmark$ (by the invariant), each such test is well-defined. Returning I^* unchanged means every test returned X, i.e., for all $\Delta_i \in I^*$, $test(I^* \setminus \{\Delta_i\}) = \mathsf{X}$. By definition, I^* is 1-minimal. □

Linear final pass when $R = 1$. The main loop visits geometrically decreasing partition sizes. Across these sizes, the number of tested blocks is bounded by a geometric series, so the coarse phase costs $O(|I|)$ oracle calls. With $R = 1$, the causal chain scan executes once at single-element granularity, also costing $O(|I|)$. Thus the overall worst-case cost is linear, but without guaranteed resolution of causal chains.

IV. PROBABILISTIC DELTA DEBUGGING (PROBDD)

A. Key Idea

ProbDD [9] was proposed as an efficiency-focused alternative to `ddmin`, accepting non-1-minimal outputs in exchange for concentrating oracle calls at productive granularities. It replaces `ddmin`'s fixed halving schedule with a learned probabilistic model. Each delta Δ_i carries a probability $p_i \in [0, 1]$ representing the belief that it is required to preserve the failure. These probabilities guide which elements to attempt removing next, and are updated after each rejected test, concentrating oracle calls on elements most likely to be removable, but providing no 1-minimality guarantee.

B. Algorithm Overview

Algorithm 6 gives the full procedure. Starting from equal initial probabilities $p_i = 0.1$, the algorithm iterates: it samples a candidate removal set, tests the reduced input, and either accepts the removal (updating I_c) or increases the probabilities of the selected deltas. The SAMPLE function (Algorithm 7) selects which deltas to remove next, sorting by ascending p_i and greedily adding to a candidate set S while the expected gain $E = |S| \cdot \prod_{s \in S}(1 - p_s)$ is increasing, stopping as soon as adding another delta would decrease E.

The UPDATEPROBS function (Algorithm 8) performs a Bayesian update when a deletion is rejected: if removing I_d caused rejection, at least one delta in I_d is required. The update ratio $r = \left(1 - \prod_{i \in I_d}(1 - p_i)\right)^{-1}$ scales each $p_i \leftarrow r \cdot p_i$; when $|I_d| = 1$, the element is definitively required and p_i set to 1.

C. Key Properties

The ProbDD algorithm has the following properties.

No 1-minimality guarantee. The probabilistic model converges when all remaining elements have probability 1, meaning the algorithm believes each is individually required. But this belief may be incorrect for elements that are part of causal chains, in which case 1-minimality may not be achieved. This is the fundamental tradeoff ProbDD accepts in exchange for its efficiency.

V. COUNTER-BASED DELTA DEBUGGING (CDD)

Zhang et al. [10] show that ProbDD's per-element probabilities behave as monotonically increasing counters (growing by a fixed factor of approximately 1.582 per round), so the Bayesian updates do not materially change the partition-size schedule. The probabilistic model is not the source of ProbDD's performance advantage over `ddmin`.

This finding motivates CDD (Counter-Based Delta Debugging): a deterministic simplification of ProbDD that discards the probabilistic model entirely and replaces it with an explicit counter r that drives the partition-size schedule directly. CDD has no per-element probability state and performs no Bayesian updates. It is structurally identical to `ddmin`^{Y}, differing only in how the partition size S is computed each round. Where `ddmin`^{Y} halves S at every step, CDD computes S from r via COMPUTE_SIZE (Algorithm 9), which derives the optimal partition size for a given estimated element-removal probability $p_r = \min(p_0 \cdot 1.582^r, 0.999)$. Zhang et al. demonstrate that CDD matches ProbDD in both effectiveness and efficiency across 76 benchmarks [10].

Algorithm 9 COMPUTE_SIZE: Partition size for round r [10]

```
function COMPUTE_SIZE(r, p_0)
    p_r ← min(p_0 × 1.582^r, 0.999)
    s* ← 1;  v* ← 0
    for s ← 1 to 9999 do
        v ← s × (1 − p_r)^s
        if v > v* then
            s* ← s;  v* ← v
        else if v < 0.99 × v* then
            break
    return s*
```

Algorithm 10 CDD: Counter-Based Delta Debugging [10]

```
function CDD(I, test, p_0 = 0.1)        ▷ 0.1 is an example.
    r ← 0
    while true do
        S ← COMPUTE_SIZE(r, p_0)
        if S ≤ 1 then break
        I ← FULL_COMPLEMENT_SCAN(I, S, test)
        r ← r + 1
    return I
```

A. Key Properties

The CDD algorithm has the following properties.

No 1-minimality guarantee. Like ProbDD, CDD terminates when $S \leq 1$ rather than after a full single-element sweep, and may therefore return a non-1-minimal output on inputs with causal chains.

Equivalent partition-size schedule to ProbDD. Zhang et al. show that CDD is exactly equivalent to ProbDD when the inputs can be partitioned without rounding, and empirically matches ProbDD's output and oracle-call count in all other cases. The two algorithms are not identical in general (CDD uses no probability state), but their partition-size schedules are functionally interchangeable. This equivalence reveals that ProbDD's advantage over `ddmin` comes entirely from its partition-size schedule, and not from its probabilistic model.

B. A Unified View

`ddmin`, `ddmin`^{Y}, `drdd`, and CDD are all instances of the same structural framework: maintain a current input I, select a partition size S according to some schedule, invoke FULL_COMPLEMENT_SCAN at that size, and decide whether and how to restart. The algorithms differ in exactly two design choices:

The partition-size schedule. `ddmin` and `ddmin`^{Y} both halve the partition size at each step ($S \leftarrow S/2$). CDD and ProbDD use the probabilistically-derived schedule $S \leftarrow$ COMPUTE_SIZE(r, p_0), producing series such as 10, 6, 4, 2, 1. This starts at a fixed height, but descends more slowly and concentrates effort at potentially more productive granularities at smaller partition sizes. `drdd` uses the same halving schedule as `ddmin`^{Y}.

The restart policy. `ddmin` restarts at the same granularity after every successful reduction. `ddmin`^{Y} restarts at the same granularity after every successful complement sweep. ProbDD and CDD perform no restarts at all. Hence, each granularity is visited exactly once. `drdd` defers all restarts to the single-element level, where they are necessary and sufficient for 1-minimality.

The partition-size schedule controls *efficiency*; the restart policy controls *minimality*: without a single-element sweep, the result may not be 1-minimal.

Within this two-axis design space, `drdd` combines the halving schedule with the minimal restart machinery needed to guarantee 1-minimality.

VI. Evaluation

Algorithms such as ProbDD and CDD make a key change relative to `ddmin`: they sacrifice the 1-minimality guarantee by performing no restarts, thereby avoiding the quadratic worst-case behavior that restarts cause. To isolate the impact of each design choice, we ask three research questions. `drdd` provides the critical baseline: by deferring restarts to the single-element level, it eliminates the intermediate restarts that cause inefficient behavior on practical inputs while retaining the 1-minimality guarantee (and the $O(n^2)$ worst case) when $R = |I|$. We use CDD as the representative of the no-restart family, since Zhang et al. show it matches ProbDD in theory and practice while being simpler and fully deterministic [10].

We note that for ProbDD and CDD, p_0 is set to the average reduction ratio observed on each subject family (see Section VI-A), which requires oracle knowledge that is unavailable in practice. The comparisons in RQ2 and RQ3 therefore favor ProbDD and CDD; the advantage of `drdd` over these baselines is thus a conservative lower bound.

RQ1. What is the performance impact of deferring restarts to the single-element level?

RQ2. Does sacrificing the 1-minimality guarantee yield a performance benefit over an algorithm that preserves it?

RQ3. What is the impact of sacrificing the 1-minimality guarantee on reduction quality?

A. Evaluation Setup

All algorithms run under identical conditions on the same failing inputs, with failure reproduction verified for each. Performance metrics are defined in Section VI-D.

B. Infrastructure

All experiments ran natively on Fedora Linux 43 (kernel 6.19, x86_64) on an AMD Ryzen AI 9 HX PRO 370 workstation with 64 GB of RAM. The harness is driven by Python 3.14 in a standard virtual environment, and each task runs sequentially on a single core.

C. Subjects

We evaluate `ddmin`^{Y}, `drdd`, ProbDD, and CDD on four subject families that expose a structure-agnostic reducer to qualitatively different input shapes: a binary media format (`ffmpeg`), a richly-nested text format (XML), a binary object format (`binutils`), and a JavaScript source format (CrashJS).

We reuse the most directly applicable prior dataset: the XML subjects are taken verbatim from the replication artifact

TABLE I
COMPARISON OF DDMIN^Y, DRDD, PROBDD, AND CDD: REDUCED SIZE AND WALL-CLOCK TIME.

	Bug ID	Input (b)	Reduced Size (b)				Wall-Clock Time (m)			
			ddmin^Y	*drdd*	*ProbDD*	*CDD*	*ddmin^Y*	*drdd*	*ProbDD*	*CDD*
ffmpeg	T-10686	154,343	**17**	**17**	**17**	20	0.11	**0.06**	0.85	0.64
	T-10688	4,980	1,245	**1,070**	1,071	**1,070**	2.71	**1.90**	3.35	**1.44**
	T-10691	3,478	**369**	**369**	552	**369**	**0.67**	1.28	1.53	**0.62**
	T-10699	67,988	**25**	**25**	30	29	**0.14**	**0.07**	0.49	0.34
	T-10701	13,713	**13,660**	**13,660**	13,713	**13,369**	16.80	**10.47**	14.10	13.71
	T-10702	2,024	**93**	94	142	103	0.21	**0.16**	0.32	**0.14**
	T-10744	4,460	**643**	**643**	1,207	**643**	1.76	**1.09**	3.23	**0.83**
	T-10745	68,054	61	**59**	2,518	1,900	3.98	**1.81**	89.64	38.04
	T-10746	468	**92**	96	96	**92**	0.17	**0.09**	0.13	0.12
	T-10747	519	**92**	**92**	**92**	**92**	0.16	**0.11**	0.14	0.12
	T-10749	3,476	408	408	**108**	408	0.63	**0.34**	1.96	0.59
	T-10754	4,984	8	**7**	8	8	**0.12**	**0.10**	0.17	**0.09**
	T-10756	542	**4**	**4**	**4**	**4**	**0.04**	**0.03**	0.06	**0.04**
	T-10758	4,995	1,625	**1,263**	**1,250**	**1,237**	79.46	**7.92**	15.30	**5.87**
	Average	23,859	1,310	1,272	1,486	1,382	7.64	1.82	9.38	4.47
XML	T-1e9bc83-1-1	1,391	**396**	401	526	523	0.60	**0.52**	0.95	0.75
	T-1e9bc83-1-2	3,974	**1,101**	1,151	1,469	1,467	1.80	**1.31**	2.55	1.99
	T-1e9bc83-1-3	5,342	**1,397**	1,413	1,845	1,888	2.75	**1.55**	3.51	2.76
	T-1e9bc83-2-1	1,939	**565**	**565**	763	763	5.74	**0.79**	1.35	1.08
	T-1e9bc83-2-2	3,997	**1,083**	1,124	1,489	1,490	12.80	**1.43**	2.61	2.03
	T-1e9bc83-2-3	5,502	**1,444**	1,450	2,014	2,015	20.28	**1.74**	3.44	2.68
	T-1e9bc83-3-1	2,015	**634**	**634**	817	817	5.13	**0.79**	1.43	1.09
	T-1e9bc83-3-2	3,835	**1,101**	**1,101**	1,490	1,492	9.06	**1.31**	2.50	1.96
	T-1e9bc83-3-3	5,755	**1,721**	1,775	2,159	2,207	18.52	**1.83**	3.83	2.88
	T-1e9bc83-4-1	3,933	**825**	886	1,226	1,225	2.64	**1.38**	2.56	2.04
	T-1e9bc83-4-2	4,014	**864**	919	1,269	1,273	2.59	**1.33**	2.55	2.08
	T-1e9bc83-4-3	5,870	**1,240**	1,314	1,854	1,856	5.49	**1.72**	3.74	2.98
	T-1e9bc83-5-1	1,983	**497**	537	734	735	4.24	**0.81**	1.31	1.07
	T-1e9bc83-5-2	4,052	**1,029**	1,075	1,475	1,476	8.85	**1.35**	2.55	1.97
	T-1e9bc83-5-3	5,497	**1,471**	1,518	2,052	2,052	15.61	**1.61**	3.35	2.58
	Average	3,940	1,025	1,058	1,412	1,419	7.74	1.30	2.55	2.00
binutils	bug-20605	21,504	**17,782**	**17,782**	**17,782**	**17,782**	1.86	**1.21**	7.03	6.51
	bug-21135	968	**640**	**640**	**640**	**640**	3.16	**0.42**	0.54	0.54
	bug-21136	3,964	**3,944**	**3,944**	**3,944**	**3,944**	0.5	0.36	**0.19**	0.17
	bug-21138	4,010	**3,944**	**3,944**	**3,944**	**3,944**	0.48	0.35	0.26	**0.24**
	bug-21139	4,011	**3,992**	**3,992**	**3,992**	**3,992**	0.64	0.25	0.21	**0.19**
	bug-21143	3,964	**3,944**	**3,944**	**3,944**	**3,944**	0.48	0.35	0.19	**0.16**
	bug-21144	3,977	**3,944**	**3,944**	**3,944**	**3,944**	0.43	0.35	0.21	**0.19**
	bug-21145	3,180	**2,680**	**2,680**	**2,680**	**2,680**	0.34	**0.24**	0.9	0.88
	bug-21409-1	1,547	**1,160**	**1,160**	**1,160**	**1,160**	0.1	**0.07**	0.66	0.65
	bug-21409-2	2,353	**1,422**	**1,422**	1,499	1,529	0.18	**0.1**	1.42	1.36
	bug-21414	683	**680**	**680**	**680**	**680**	0.06	0.05	**0.03**	**0.03**
	bug-30886	16,880	**12,321**	**12,321**	**12,321**	**12,321**	1.74	**1.32**	7.97	7.64
	Average	5,587	4,704	4,704	4,711	4,713	0.83	0.42	1.63	1.55
crashjs	lodash-1	1,827	**537**	542	726	705	0.50	**0.04**	0.06	**0.04**
	lodash-2	613	**401**	404	404	404	0.25	0.03	**0.02**	**0.02**
	lodash-3	1,180	**744**	747	755	816	1.24	0.07	0.04	**0.03**
	lodash-4	624	**396**	397	400	399	0.24	0.03	**0.02**	**0.02**
	lodash-5	2,356	**1,295**	1,298	1,406	1,541	5.21	0.13	0.08	**0.07**
	lodash-6	807	**171**	176	264	241	0.11	**0.02**	**0.02**	**0.02**
	lodash-7	1,012	369	**361**	464	480	0.37	0.04	0.03	**0.02**
	lodash-8	1,024	**256**	259	348	314	0.13	**0.02**	0.03	**0.02**
	lodash-9	444	226	**217**	292	301	0.11	0.02	0.02	**0.01**
	lodash-10	830	**352**	355	409	413	0.31	0.04	0.03	**0.02**
	lodash-11	1,090	**488**	494	595	597	0.59	0.04	**0.03**	**0.03**
	Average	1,073	476	477	551	565	0.82	0.04	0.03	0.03

bold *indicates the best value if there is a tie.*
indicates the best unique *value among ddmin^Y, drdd, ProbDD and CDD*

TABLE II
COMPARISON OF DDMIN^Y, DRDD, PROBDD, AND CDD: REDUCED SIZE AND ORACLE CALLS.

	Bug ID	Input (b)	Reduced Size (% of Input)				# Oracle Calls			
			ddmin^Y	*drdd*	*ProbDD*	*CDD*	*ddmin^Y*	*drdd*	*ProbDD*	*CDD*
ffmpeg	T-10686	154,343	**0.0**	**0.0**	**0.0**	**0.0**	266	**141**	1,756	1,649
	T-10688	4,980	25.0	**21.5**	**21.5**	**21.5**	5,181	3,216	5,932	**2,402**
	T-10691	3,478	**10.6**	**10.6**	15.9	**10.6**	**1,292**	2,543	3,185	**1,203**
	T-10699	67,988	**0.0**	**0.0**	**0.0**	**0.0**	326	**172**	954	802
	T-10701	13,713	99.6	99.6	100.0	**97.5**	36,408	**22,588**	29,760	29,314
	T-10702	2,024	**4.6**	**4.6**	7.0	5.1	434	**319**	641	**287**
	T-10744	4,460	**14.4**	**14.4**	27.1	**14.4**	3,149	1,952	5,221	**1,456**
	T-10745	68,054	**0.1**	**0.1**	3.7	2.8	788	**345**	12,990	4,868
	T-10746	468	**19.7**	20.5	20.5	**19.7**	308	**163**	220	212
	T-10747	519	**17.7**	**17.7**	**17.7**	**17.7**	280	**193**	217	211
	T-10749	3,476	11.7	11.7	**3.1**	11.7	1,335	**688**	1,075	927
	T-10754	4,984	**0.2**	**0.1**	**0.2**	**0.2**	**44**	51	125	78
	T-10756	542	**0.7**	**0.7**	**0.7**	**0.7**	**23**	24	51	26
	T-10758	4,995	32.5	**25.3**	**25.0**	**24.8**	22,712	**3,793**	6,551	**2,802**
	Average	23,859	16.9	16.2	17.3	16.2	5,182	2,585	4,906	3,303
XML	T-1e9bc83-1-1	1,391	**28.5**	28.8	37.8	37.6	32,567	**1,618**	1,766	**1,468**
	T-1e9bc83-1-2	3,974	**27.7**	29.0	37.0	36.9	204,368	**4,228**	4,980	**4,146**
	T-1e9bc83-1-3	5,342	**26.2**	26.5	34.5	35.3	354,262	**5,660**	6,509	**5,480**
	T-1e9bc83-2-1	1,939	**29.1**	**29.1**	39.4	39.4	58,843	**2,072**	2,505	2,116
	T-1e9bc83-2-2	3,997	**27.1**	28.1	37.3	37.3	232,469	**4,197**	5,066	4,199
	T-1e9bc83-2-3	5,502	**26.2**	26.4	36.6	36.6	482,234	**5,902**	6,811	**5,739**
	T-1e9bc83-3-1	2,015	**31.5**	**31.5**	40.5	40.5	69,237	**2,241**	2,672	**2,214**
	T-1e9bc83-3-2	3,835	**28.7**	**28.7**	38.9	38.9	199,297	**3,978**	4,914	4,130
	T-1e9bc83-3-3	5,755	**29.9**	30.8	37.5	38.3	574,863	**6,850**	7,339	**6,081**
	T-1e9bc83-4-1	3,933	**21.0**	22.5	31.2	31.1	163,489	**3,594**	4,496	3,763
	T-1e9bc83-4-2	4,014	**21.5**	22.9	31.6	31.7	162,109	**3,631**	4,549	3,886
	T-1e9bc83-4-3	5,870	**21.1**	22.4	31.6	31.6	425,731	**5,529**	6,786	5,657
	T-1e9bc83-5-1	1,983	**25.1**	27.1	37.0	37.1	45,155	**2,037**	2,473	2,126
	T-1e9bc83-5-2	4,052	**25.4**	26.5	36.4	36.4	184,590	**4,027**	5,053	4,212
	T-1e9bc83-5-3	5,497	**26.8**	27.6	37.3	37.3	459,943	**6,056**	6,961	**5,855**
	Average	3,940	26.4	27.2	36.3	36.4	243,277	4,108	4,859	4,071
binutils	bug-20605	21,504	**82.7**	**82.7**	**82.7**	**82.7**	51,626	33,710	**21,504**	**21,504**
	bug-21135	968	**66.1**	**66.1**	**66.1**	**66.1**	85,119	2,042	**968**	**968**
	bug-21136	3,964	**99.5**	**99.5**	**99.5**	**99.5**	14,238	10,294	**3,964**	**3,964**
	bug-21138	4,010	**98.4**	**98.4**	**98.4**	**98.4**	14,285	10,290	**4,010**	**4,010**
	bug-21139	4,011	**99.5**	**99.5**	**99.5**	**99.5**	16,551	6,422	**4,011**	**4,011**
	bug-21143	3,964	**99.5**	**99.5**	**99.5**	**99.5**	14,238	10,294	**3,964**	**3,964**
	bug-21144	3,977	**99.2**	**99.2**	**99.2**	**99.2**	12,659	10,289	**3,977**	**3,977**
	bug-21145	3,180	**84.3**	**84.3**	**84.3**	**84.3**	10,218	7,156	**3,180**	**3,180**
	bug-21409-1	1,547	**75.0**	**75.0**	**75.0**	**75.0**	3,105	1,940	**1,547**	**1,547**
	bug-21409-2	2,353	**60.4**	**60.4**	63.7	65.0	4,891	2,812	**2,353**	**2,353**
	bug-21414	683	**99.6**	**99.6**	**99.6**	**99.6**	1,793	1,290	**683**	**683**
	bug-30886	16,880	**73.0**	**73.0**	**73.0**	**73.0**	48,635	36,970	**16,880**	**16,880**
	Average	5,587	86.4	86.4	86.7	86.8	23,113	11,126	5,587	5,587
crashjs	lodash-1	1,827	**29.4**	29.7	39.7	38.6	40,178	2,318	2,146	**1,954**
	lodash-2	613	**65.4**	65.9	65.9	65.9	20,069	1,417	845	**781**
	lodash-3	1,180	**63.1**	63.3	64.0	69.2	101,563	3,553	1,642	**1,560**
	lodash-4	624	**63.5**	63.6	64.1	63.9	18,750	1,396	862	**780**
	lodash-5	2,356	**55.0**	55.1	59.7	65.4	380,685	7,853	3,196	**3,068**
	lodash-6	807	**21.2**	21.8	32.7	29.9	8,838	853	894	**806**
	lodash-7	1,012	36.5	**35.7**	45.8	47.4	30,989	2,023	1,266	**1,122**
	lodash-8	1,024	**25.0**	25.3	34.0	30.7	9,726	1,223	1,150	**986**
	lodash-9	444	50.9	**48.9**	65.8	67.8	8,493	982	624	**574**
	lodash-10	830	**42.4**	42.8	49.3	49.8	25,862	2,343	1,053	**939**
	lodash-11	1,090	**44.8**	45.3	54.6	54.8	46,656	2,408	1,429	**1,283**
	Average	1,073	45.2	45.2	52.3	53.0	62,892	2,397	1,373	1,259

bold *indicates the best value if there is a tie.*
indicates the best unique *value among ddmin^Y, drdd, ProbDD and CDD*

of Zhang et al. [10]. The remaining ProbDD and CDD evaluations in that work pair these algorithms with hierarchical (structure-aware) reducers on text-format inputs; reusing those subjects for a structure-agnostic, byte-level comparison would change the reduction task and conflate algorithmic differences with format-specific assumptions.

For ProbDD and CDD we follow Wang et al. [9] and set the prior p_0 to the family's average *reduction ratio* m/n, where m is the reduced size and n the input size. This gives ProbDD and CDD oracle knowledge of each family's reducibility, knowledge unavailable in practice for unknown formats. The comparisons that follow therefore favor these baselines; the advantage of `drdd` is a conservative lower bound. Sensitivity to a non-oracle choice of p_0 is examined in Section VIII.

a) `ffmpeg` bugs.: 14 binary inputs from `ffmpeg` trac bug reports that trigger AddressSanitizer reports when run through a specific filter on the pinned commit. Reducibility is sharply bimodal: some inputs collapse to a handful of bytes while others remain near full size, exercising the algorithms across both regimes. The reduction ratio for this class is highly variable, but often approaches near-complete reduction, so we use $p_0 = 0.01$.

b) XML bugs.: 15 inputs from 5 XQuery output-discrepancy cases in the replication artifact of Zhang et al. [10], with three pre-shrunk size variants per case. Each case is a discriminating XQuery evaluated against two BaseX commits (good `1e9bc83`, buggy `816b386`); the predicate accepts an input iff the buggy commit's output disagrees with both the good commit and a Saxon reference. XML's textual nesting makes complement-based reduction reliably productive, and each accepted complement opens further reductions at the same granularity, precisely the regime where `ddmin`'s restart-after-success policy is most expensive. The average reduction ratio is 0.26, giving $p_0 = 0.26$.

c) `binutils` bugs.: 12 ELF and object-file inputs that crash a single `binutils` tool (`readelf`, `objdump`, `objcopy`, or `nm`), drawn from bugzilla bugs and rebuilt against four pinned upstream commits. The dense internal layout (symbol tables, section headers, relocation entries) invalidates most byte-level complements, so reductions are rare and the algorithms converge to nearly identical outputs; the family tests overhead rather than minimization depth. The average reduction ratio is 0.86, giving $p_0 = 0.86$.

d) `crashjs` bugs.: 11 `mocha` test files from the `syntest-collected/lodash` sub-corpus of the CrashJS dataset [11]. Each input is a `Syntest`-generated reproducer that triggers a `TypeError` deep inside an instrumented `lodash` build; the predicate matches reproductions on the triple (error class, error message, top-most `lodash` frame). Inputs are small JavaScript source files (444 B–2 356 B); most byte mutations are rejected as syntax errors before the predicate runs, but the signature-matching predicate is permissive about test scaffolding, so reductions reach moderate depth, between XML's deep reductions and `binutils`'s near-irreducibility. The average reduction ratio is 0.45, giving $p_0 = 0.45$.

D. Evaluation Metrics

We evaluate performance using three primary metrics.

Let S^i be the input size, and S^o be the output size. T denotes the wall clock time, and O_{method} is the number of oracle calls for *method*.

- *reduction quality:* Measures how much a technique reduces the input relative to $\texttt{ddmin}^Y$'s reduction, defined as $((S^i - S^o_{\text{method}})/(S^i - S^o_{\texttt{ddmin}^Y})) \times 100\%$. A value of 100 indicates the method achieved the same reduction as $\texttt{ddmin}^Y$; values below 100 indicate the method removed fewer bytes than $\texttt{ddmin}^Y$; values above 100 indicate the method removed more bytes than $\texttt{ddmin}^Y$. Higher values correspond to better performance.
- *speedup:* Captures relative runtime improvement in wall-clock time, computed as $T_{\texttt{ddmin}^Y}/T_{\text{method}}$. Higher values correspond to better performance.
- *oracle call ratio:* Captures relative oracle usage, calculated as $(O_{\text{method}}/O_{\texttt{ddmin}^Y}) \times 100\%$. Lower values correspond to better performance (fewer oracle calls).

These metrics allow comparing algorithms in terms of reduction effectiveness and computational efficiency.

VII. Experimental Results

Table I presents the comparative results of running all four algorithms on `ffmpeg`, XML, `binutils`, and `crashjs` bugs, measured in wall-clock time (minutes). While wall-clock time is a useful metric, it is affected by the execution time of the oracle, which may skew comparisons across different subjects. Hence, we also report results in terms of oracle calls in Table II, which directly reflects algorithmic efficiency independent of oracle execution time.

Table III presents the relative reduction achieved and the number of oracle calls in terms of that required by $\texttt{ddmin}^Y$, for `ffmpeg`, XML, `binutils`, and `crashjs` bugs.

RQ1: Performance impact of deferring restarts

To answer RQ1, we compared $\texttt{ddmin}^Y$ (no defer) to `drdd` (deferred restart). The comparison in terms of wall-clock time is presented in Table I, and in terms of oracle calls in Table II.

As shown in Table II, `drdd` uses 2,585 oracle calls on average on `ffmpeg` versus 5,182 for $\texttt{ddmin}^Y$ (i.e., 50% on average), 4,108 versus 243,277 on XML (i.e., 1.7% on average), 11,126 versus 23,113 on `binutils` (i.e., 48% on average), and 2,397 versus 62,892 on `crashjs` (i.e., 3.8% on average). The wall-clock speedup (Table I) mirrors this: `drdd` is 4.2× faster than $\texttt{ddmin}^Y$ on `ffmpeg`, 6.0× faster on XML, 2.0× faster on `binutils`, and 19× faster on `crashjs`. Reduction quality is largely unchanged: `drdd`'s average output on `ffmpeg` is 1,272 b versus 1,310 b for $\texttt{ddmin}^Y$ (i.e., 101.02% reduction quality), 1,058 versus 1,025 on XML (i.e., 98.92% reduction quality, a modest ≈1% gap discussed in Section VIII), 4,704 b versus 4,704 b on `binutils` (i.e., 100.00%), and 477 b versus 476 b on `crashjs` (i.e., 99.94%) (Table III). On `binutils`, the savings are moderate: the binary format structure limits the number of successful complement tests, so restart-induced

TABLE III
COMPARISON OF DDMIN^Y, DRDD, PROBDD, AND CDD: REDUCTION QUALITY AND ORACLE CALLS RELATIVE TO DDMIN^Y.

	Bug ID	Input (b)	Reduction Quality (% of ddmin^Y)				Oracle Call Ratio (% of ddmin^Y)			
			ddmin^Y	*drdd*	*ProbDD*	*CDD*	*ddmin^Y*	*drdd*	*ProbDD*	*CDD*
ffmpeg	T-10686	154,343	**100.00**	100.00	100.00	100.00	**100.00**	**53.01**	660.15	619.92
	T-10688	4,980	100.00	104.69	104.66	104.69	100.00	62.07	114.50	**46.36**
	T-10691	3,478	**100.00**	100.00	94.11	100.00	100.00	196.83	246.52	**93.11**
	T-10699	67,988	**100.00**	100.00	99.99	99.99	100.00	**52.76**	292.64	246.01
	T-10701	13,713	**100.00**	**100.00**	0.00	**649.06**	100.00	**62.04**	81.74	80.52
	T-10702	2,024	**100.00**	99.95	97.46	99.48	100.00	73.50	147.70	**66.13**
	T-10744	4,460	**100.00**	100.00	85.22	100.00	100.00	61.99	165.80	**46.24**
	T-10745	68,054	**100.00**	100.00	96.39	97.30	100.00	**43.78**	1,648.48	617.77
	T-10746	468	**100.00**	98.94	98.94	100.00	100.00	**52.92**	71.43	68.83
	T-10747	519	**100.00**	100.00	100.00	100.00	100.00	**68.93**	77.50	75.36
	T-10749	3,476	**100.00**	**100.00**	**109.78**	**100.00**	100.00	**51.54**	80.52	69.44
	T-10754	4,984	**100.00**	**100.02**	**100.00**	**100.00**	**100.00**	115.91	284.09	177.27
	T-10756	542	**100.00**	100.00	100.00	100.00	**100.00**	104.35	221.74	113.04
	T-10758	4,995	100.00	110.74	111.13	**111.51**	100.00	16.70	28.84	**12.34**
	Average	23,859	100.00	101.02	92.69	140.14	100.00	72.59	294.40	166.60
XML	T-1e9bc83-1-1	1,391	**100.00**	99.50	86.93	87.24	100.00	4.97	5.42	**4.51**
	T-1e9bc83-1-2	3,974	**100.00**	98.26	87.19	87.26	100.00	2.07	2.44	**2.03**
	T-1e9bc83-1-3	5,342	**100.00**	99.59	88.64	87.55	100.00	1.60	1.84	**1.55**
	T-1e9bc83-2-1	1,939	**100.00**	100.00	85.59	85.59	100.00	**3.52**	4.26	3.60
	T-1e9bc83-2-2	3,997	**100.00**	98.59	86.07	86.03	100.00	1.81	2.18	1.81
	T-1e9bc83-2-3	5,502	**100.00**	99.85	85.95	85.93	100.00	1.22	1.41	**1.19**
	T-1e9bc83-3-1	2,015	**100.00**	100.00	86.75	86.75	100.00	3.24	3.86	**3.20**
	T-1e9bc83-3-2	3,835	**100.00**	100.00	85.77	85.70	100.00	**2.00**	2.47	2.07
	T-1e9bc83-3-3	5,755	**100.00**	98.66	89.14	87.95	100.00	1.19	1.28	**1.06**
	T-1e9bc83-4-1	3,933	**100.00**	98.04	87.10	87.13	100.00	**2.20**	2.75	2.30
	T-1e9bc83-4-2	4,014	**100.00**	98.25	87.14	87.02	100.00	**2.24**	2.81	2.40
	T-1e9bc83-4-3	5,870	**100.00**	98.40	86.74	86.70	100.00	**1.30**	1.59	1.33
	T-1e9bc83-5-1	1,983	**100.00**	97.31	84.05	83.98	100.00	**4.51**	5.48	4.71
	T-1e9bc83-5-2	4,052	**100.00**	98.48	85.25	85.21	100.00	**2.18**	2.74	2.28
	T-1e9bc83-5-3	5,497	**100.00**	98.83	85.57	85.57	100.00	1.32	1.51	**1.27**
	Average	3,940	100.00	98.92	86.53	86.37	100.00	2.36	2.80	2.35
binutils	bug-20605	21,504	**100.00**	100.00	100.00	100.00	100.00	65.30	**41.65**	**41.65**
	bug-21135	968	**100.00**	100.00	100.00	100.00	100.00	2.40	**1.14**	**1.14**
	bug-21136	3,964	**100.00**	100.00	100.00	100.00	100.00	72.30	**27.84**	**27.84**
	bug-21138	4,010	**100.00**	100.00	100.00	100.00	100.00	72.03	**28.07**	**28.07**
	bug-21139	4,011	**100.00**	100.00	100.00	100.00	100.00	38.80	**24.23**	**24.23**
	bug-21143	3,964	**100.00**	100.00	100.00	100.00	100.00	72.30	**27.84**	**27.84**
	bug-21144	3,977	**100.00**	100.00	100.00	100.00	100.00	81.28	**31.42**	**31.42**
	bug-21145	3,180	**100.00**	100.00	100.00	100.00	100.00	70.03	**31.12**	**31.12**
	bug-21409-1	1,547	**100.00**	100.00	100.00	100.00	100.00	62.48	**49.82**	**49.82**
	bug-21409-2	2,353	**100.00**	100.00	91.73	88.51	100.00	57.49	**48.11**	**48.11**
	bug-21414	683	**100.00**	100.00	100.00	100.00	100.00	71.95	**38.09**	**38.09**
	bug-30886	16,880	**100.00**	100.00	100.00	100.00	100.00	76.02	**34.71**	**34.71**
	Average	5,587	100.00	100.00	99.31	99.04	100.00	61.87	32.00	32.00
crashjs	lodash-1	1,827	**100.00**	99.61	85.35	86.98	100.00	5.77	5.34	**4.86**
	lodash-2	613	**100.00**	98.58	98.58	98.58	100.00	7.06	4.21	**3.89**
	lodash-3	1,180	**100.00**	99.31	97.48	83.49	100.00	3.50	1.62	**1.54**
	lodash-4	624	**100.00**	99.56	98.25	98.68	100.00	7.45	4.60	**4.16**
	lodash-5	2,356	**100.00**	99.72	89.54	76.81	100.00	2.06	0.84	**0.81**
	lodash-6	807	**100.00**	99.21	85.38	88.99	100.00	9.65	10.12	**9.12**
	lodash-7	1,012	100.00	**101.24**	85.23	82.74	100.00	6.53	4.09	**3.62**
	lodash-8	1,024	**100.00**	99.61	88.02	92.45	100.00	12.57	11.82	**10.14**
	lodash-9	444	100.00	**104.13**	69.72	65.60	100.00	11.56	7.35	**6.76**
	lodash-10	830	**100.00**	99.37	88.08	87.24	100.00	9.06	4.07	**3.63**
	lodash-11	1,090	**100.00**	99.00	82.23	81.89	100.00	5.16	3.06	**2.75**
	Average	1,073	100.00	99.94	87.99	85.77	100.00	7.31	5.19	4.66

bold *indicates the best value if there is a tie.*
indicates the best unique *value among* *ddmin^Y*, *drdd*, *ProbDD and CDD*
indicates the best unique *value among* *drdd*, *ProbDD and CDD*

TABLE IV
POST-HOC 1-MINIMALITY VERIFICATION OF PROBDD AND CDD OUTPUTS. *S* = BYTES REMOVED BY THE VERIFIER (NON-ZERO ⇒ NOT 1-MINIMAL). *Gap* = BYTES STILL ABOVE DRDD AFTER VERIFICATION.

	Bug ID	drdd (b)	ProbDD		CDD	
			S	*Gap*	*S*	*Gap*
binutils	bug-21414	680	0	0	0	0
	bug-21135	640	0	0	0	0
	bug-21409-2	1,422	81	0	80	0
	bug-21136	3,944	0	0	0	0
crashjs	lodash-9	217	3	72	12	72
	lodash-6	176	46	42	22	43
	lodash-1	542	64	120	44	119
	lodash-3	747	16	−8	31	38
	lodash-5	1,298	68	40	194	49
ffmpeg	T-10686	17	0	0	3	0
	T-10702	94	7	816	0	9
	T-10744	643	0	753	0	0
XML	T-1e9bc83-1-1	401	0	123	0	122
	T-1e9bc83-3-1	634	0	183	0	183
	T-1e9bc83-1-2	1,151	0	317	0	316
1-minimal (of 15)			*8 / 15*		*8 / 15*	

overhead is less severe than on XML. On crashjs, the JavaScript source format presents many invalid byte-level complements, yet the large number of successful reductions per input means restarts in ddmin^Y are triggered very frequently, making the savings from deferral the most dramatic in wall-clock speedup across all four subjects.

drdd uses 50% of ddmin^Y's oracle calls on ffmpeg (4.2× speedup), 1.7% on XML (6.0× speedup), 48% on binutils (2.0× speedup), and 3.8% on crashjs (19× speedup), while preserving 1-minimality and matching reduction quality.

RQ2: Performance benefit of sacrificing 1-minimality

To answer RQ2, we compare drdd against both ProbDD and CDD using Table II.

As shown in Table II, on ffmpeg, ProbDD uses 4,906 oracle calls on average versus 2,585 for drdd (i.e., 1.9× more), and is 5.2× slower in wall-clock time (Table I). On XML, ProbDD uses 4,859 oracle calls on average versus 4,108 for drdd (i.e., 1.2× more), and is 2.0× slower. On binutils, ProbDD uses 5,587 oracle calls on average versus 11,126 for drdd (i.e., approximately half), but is substantially slower in wall-clock time (1.63 m versus 0.42 m for drdd). On crashjs, ProbDD uses 1,373 oracle calls on average versus 2,397 for drdd (i.e., 0.57×), and CDD uses 1,259 (i.e., 0.53×), both fewer than drdd, yet wall-clock time is indistinguishable (0.03 m for both versus 0.04 m for drdd). CDD shows the same pattern as ProbDD on the other subjects: on ffmpeg it uses 3,303 oracle calls on average, on XML the counts are nearly identical (4,071), and on binutils it uses 5,587 oracle calls, matching ProbDD and substantially fewer than drdd. Sacrificing 1-minimality therefore yields no oracle-call benefit over drdd on ffmpeg or XML, regardless of whether the probabilistic model is retained (ProbDD) or discarded (CDD), but does yield a benefit on binutils and crashjs. On both subjects, the probabilistic schedule's tendency to start with small partitions (guided by a high p_0) proves effective at avoiding oracle calls on large partitions unlikely to succeed. However, this comes at the cost of non-1-minimal outputs, as the quality analysis in RQ3 shows.

Sacrificing 1-minimality provides no meaningful oracle-call benefit on ffmpeg or XML, and on binutils and crashjs trades fewer oracle calls for consistently larger outputs.

RQ3: Impact of sacrificing 1-minimality on reduction quality

Table III shows that ProbDD and CDD often produce larger outputs than ddmin^Y, although individual cases can tie or even improve due to different greedy search paths. To distinguish *why* the gap arises, we ran the single-element fixed-point verifier against each competitor output (Table IV). The results reveal two separate mechanisms.

Mechanism 1: genuine non-1-minimality. On crashjs all five verified subjects are non-1-minimal for both ProbDD and CDD. On binutils, bug-21409-2 is also non-1-minimal: JavaScript source files contain syntactic causal chains which neither ProbDD nor CDD resolves without a fixed-point scan. The fact that CDD, which discards the probabilistic model entirely, suffers the same loss as ProbDD confirms that the degradation stems from the missing restart, not from the probabilistic schedule.

Mechanism 2: a different (worse) 1-minimal fixed point. On XML, the verifier finds zero shortfall for all three verified subjects: the competitor outputs are already 1-minimal. Yet they sit 122–317 b above drdd's output. As we will note in Section VIII, multiple 1-minimal fixed points exist, and the restart-free schedule arrives at a different one than drdd's deferred-restart path.

On ffmpeg, ProbDD's average output is 1,486 versus 1,310 for ddmin^Y (i.e., 92.69% reduction quality), a modest degradation driven mainly by subjects where ProbDD fails to remove elements (e.g., T-10745 and T-10744 in Table II). CDD shows the same pattern on ffmpeg, with its average inflated by T-10701 where it happens to reach a better reduction than ddmin^Y. On XML the degradation is more pronounced: ProbDD's average output is 1,412 versus 1,025 for ddmin^Y (i.e., 86.53% reduction quality), and CDD's average output is 1,419 (i.e., 86.37%), both substantially below drdd's 1,058 (i.e., 98.92%). Verification confirms this gap is entirely due to a worse fixed point, not to non-1-minimality. On binutils, the degradation is modest overall: ProbDD's average output is 4,711 versus 4,704 for ddmin^Y (i.e., 99.31% reduction quality), and CDD's average output is 4,713, both close to ddmin^Y. On crashjs, the degradation is clearly visible: ProbDD's average output is 551 b versus 476 b for ddmin^Y (i.e., 87.99% reduction quality), and CDD's average output is 565 b (i.e., 85.77%). Verification confirms all five crashjs subjects are non-1-minimal.

Quality degradation in ProbDD and CDD arises from two mechanisms: non-1-minimality on `crashjs` and `binutils` and alternate 1-minimal fixed-point in XML.

VIII. Discussion

Comparing ProbDD to `drdd`, ProbDD uses more oracle calls than `drdd` on both `ffmpeg` and XML (1.9× more on `ffmpeg` and 1.2× more on XML) while producing larger outputs. On `binutils`, ProbDD uses approximately half the oracle calls of `drdd` (5,587 vs. 11,126), but is substantially slower in wall-clock time (1.63 m vs. 0.42 m). On `crashjs`, ProbDD and CDD also use fewer oracle calls than `drdd` (1,373 and 1,259 vs. 2,397), and wall-clock time is practically identical across all four algorithms at 0.03–0.04 m. Sacrificing 1-minimality therefore yields no meaningful oracle-call benefit over `drdd` for ProbDD or CDD on `ffmpeg` or XML. On `binutils` and `crashjs`, both ProbDD and CDD use fewer oracle calls than `drdd`, but this comes at the cost of larger outputs.

`drdd` uses fewer oracle calls than ProbDD on average on `ffmpeg` and XML, while guaranteeing 1-minimality and producing smaller or equal outputs on all four subjects. On `binutils` and `crashjs`, ProbDD and CDD use fewer oracle calls than `drdd`, but produce consistently larger outputs.

A. Impact of the subject on achieved reduction

The four subjects span a spectrum of restart-induced overhead. On `ffmpeg`, the binary format prevents most complement tests from succeeding, so restarts are rarely triggered and the savings from deferral are modest. On XML, the textual structure allows many more complements to succeed. `ddmin`^{Y} removes one chunk per complement scan and then restarts from the top, paying the descent cost again for each removal; `drdd` removes all removable chunks at a given partition size in a single pass, eliminating those repeated descents (e.g., 32,567 oracle calls for `ddmin`^{Y} versus 1,618 for `drdd` on T-1e9bc83-1-1). `binutils` sits between the two: fewer successful complements than XML but more than `ffmpeg`, yielding roughly a factor-of-two saving. `crashjs` represents the extreme end of restart-induced overhead: the JavaScript source format allows many reductions to succeed, and most inputs reduce substantially, so `ddmin`^{Y} triggers restarts very frequently. `drdd` eliminates nearly all of this overhead, using only 3.8% of `ddmin`^{Y}'s oracle calls, the largest relative saving across all four subjects after XML's 1.7%.

B. Why Restarts At Every Partition Sometimes Improve Reduction Quality

Our observations raise a question: why does `ddmin`^{Y} find a *better* reduction than `drdd` in many cases?

Several effects explain this. First, structured inputs contain elements that can only be removed successfully as a complete unit. In XML, a complement that splits a tag pair such as `<div>···</div>` produces markup that the oracle rejects. Numerous tests on differing boundaries increase the chance of successfully covering such pairs in one reduction call. Second, 1-minimality does not imply global minimality: multiple 1-minimal fixed points may exist, and the restart-induced search path may lead `ddmin`^{Y} to a smaller one. In short, increasing the number of oracle calls raises the chance of reaching a smaller fixed point via a different search trajectory. Table IV confirms this directly: on XML, verifier shortfall is zero, so the gap is between two different 1-minimal fixed points, not missing elements.

C. Why `ddmin`^{Y} Is Sometimes Not the Best

All algorithms in the `ddmin` family are greedy: each step accepts the first successful reduction, so the order in which complements are tested determines which 1-minimal fixed point is reached. Different schedules or restart policies follow different paths and may arrive at a different, sometimes smaller, fixed point. This is visible in T-10701 (Table II), where CDD's schedule finds a better reduction (13,369 b) than both `ddmin`^{Y} (13,660 b) and ProbDD (no reduction at all), confirming that the effect is caused by exploration order, not by the probabilistic model.

TABLE V
Effect of restart budget R on oracle calls, output size, and 1-minimality, averaged over 15 subjects (same as Table IV).

R	Oracle calls	Output (b)	S (b)	% 1-min
1	1,829	857.5	17.1	60
2	2,546	841.1	0.7	93
4	2,833	840.4	0	100
8	2,833	840.4	0	100
$\|I\|$	2,833	840.4	0	100

D. The Role of the Restart Budget R

Table V sweeps $R \in \{1, 2, 4, 8, |I|\}$ over the 15 subjects also used for the verification study. The tradeoff collapses early: $R = 4$ achieves 100% 1-minimality across all subjects, and $R = 8$ and $R = |I|$ add nothing: the fixed-point break fires and no further scans run.

Zhang et al. [10] suggest using CDD in a fixed-point loop as an alternative. This is essentially similar to `ddmin`^{Y}, and ignores the fact that 1-minimality need not be checked at every partition. `drdd`, by contrast, bounds the number of single-element scans by R, giving a guaranteed $O(n \cdot R)$ worst case that the caller controls explicitly.

E. Lack of prior belief

Both CDD and ProbDD rely on p_0 encoding a prior belief about input reducibility, and their performance is highly sensitive to this choice. In our evaluation, we set p_0 to the average reduction ratio observed on the same subjects, an oracle value that is impossible to know in advance for an unknown format.

The impact is substantial, and can be seen by examining `compute_size(round,` p_0`)`, which determines the initial partition size. On `binutils`, with an oracle prior of $p_0 = 0.8$, `compute_size(0,` p_0`)` returns 1. In effect, the algorithm degenerates into a linear scan. Under this setting, ProbDD and CDD require only 5,587 oracle calls on average — roughly half of `drdd`'s 11,126.

However, this advantage shrinks dramatically under a more realistic $p_0 = 0.1$. Oracle calls rise to 11,228 for ProbDD and 10,288 for CDD, both now close to `drdd`'s 11,126. The gap in Table II therefore reflects best-case. In practice, ProbDD and CDD offer no consistent advantage over `drdd`.

`drdd` requires no prior knowledge, replacing p_0 sensitivity with explicit bounds S^{max} and S^{min}.

IX. Related Work

Delta debugging was introduced by Zeller and Hildebrandt [6] for structure-agnostic input minimization.

The most recent work closely related to ours is by Zhang et al. [10], a theoretical and empirical study that seeks to better understand the ProbDD algorithm. Zhang et al. also introduce CDD, a deterministic simplification of ProbDD that replaces the probabilistic model with an explicit counter, achieving the same effectiveness and efficiency [10].

Misherghi and Su [12] introduced Hierarchical Delta Debugging (HDD). Subsequent work modernized HDD [13] and introduced HDDr [14], reducing both output size and reduction time. Grammar-based reducers such as Perses [15] and CHISEL [16] further exploit input structure to improve reducibility, but require a format specification and a parseable input, conditions that frequently fail for fuzzer-generated or malformed binary inputs.

Parallelization [17], generalized split operators [18], and relaxed minimality guarantees [9] are complementary approaches. Vince [19] introduced DDMIN*, a fixed-point iteration converging to a better local optimum, and is closely related to the restart mechanism. Backtracking becomes more expensive than linear scanning once the shrinkage is 15% [20], motivating efficient restart policies like `drdd`.

X. Threats to Validity

Internal Validity. Two non-determinism sources: First, ProbDD alone is stochastic (`ddmin`^{Y}, `drdd`, and CDD are deterministic), and we report a single ProbDD run per subject. This does not favor `drdd`: seed sensitivity is minor on `binutils`, `crashjs`, and XML but pronounced on `ffmpeg`'s highly reducible inputs, where `ticket-10702` spans 62–917 b (15×) across seeds 0–9. A single ProbDD run is thus unreliable exactly where ProbDD looks strongest; real use would demand repeated runs, a cost our oracle counts omit. This stochasticity does not affect our conclusions: the 1-minimality results are structural, and ProbDD's family-level aggregates shift by under 1.5% across seeds. Second, ASLR perturbs the `binutils` oracle: `bug-21409-2` shows ±12 b variance in competitor start size, as its SIGSEGV-based oracle faults a borderline out-of-bounds access only in some `objdump` layouts. Neither affects our conclusions: `drdd` and `ddmin`^{Y} stay consistent, and the verifier converges to 1422 b.
External Validity. We acknowledge that the limited diversity in subjects can limit the generality of our findings. As a mitigation, we note that we examined four different formats, which cover a representative range of the structure types that challenge structure-agnostic reducers; even if generalizability beyond these formats cannot be established, our findings remain practically useful. The per-family subject counts are small (3–5 per family), which limits statistical confidence in family-level conclusions.
Construct Validity. We rely on standard and well-established measurement techniques: wall-clock time, reduced input size, and the number of oracle calls.

XI. Conclusion

Delta debugging is widely used for minimizing failure-inducing inputs, but `ddmin`'s quadratic worst-case cost limits its practicality. We show that this cost arises in practice from unnecessary restarts at intermediate granularities, and that restarts are only required at the single-element level to preserve 1-minimality. Based on this insight, we introduce `drdd`, which defers restarts to the final level, retaining the worst-case bound while reducing redundant work on practical inputs.

Across `ffmpeg`, XML, `binutils`, and `crashjs` inputs, `drdd` matches `ddmin`^{Y}'s reduction quality while using substantially fewer oracle calls. It uses just 1.7% oracle calls on XML and 50% on `ffmpeg`. Unlike ProbDD and CDD, it achieves these gains without sacrificing 1-minimality or requiring prior tuning. Overall, `drdd` provides a simple, deterministic drop-in replacement for `ddmin` that makes structure-agnostic input reduction more efficient and robust in practice.

References

[1] W. Choi, K. Sen, G. Necula, and W. Wang, "DetReduce: Minimizing Android GUI test suites for regression testing," in *ICSE*, 2018.
[2] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *PLDI*, 2013.
[3] A. Zeller, "Isolating failure-inducing input," *IEEE TSE*, 2001.
[4] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *ISSREW*, 2018.
[5] C. Zhang, A. Groce, and M. A. Alipour, "Using test case reduction and prioritization to improve symbolic execution," in *ACM ISSTA*, 2014.
[6] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE TSE*, vol. 28, no. 2, pp. 183–200, 2002.
[7] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.
[8] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Reducing failure-inducing inputs," in *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. [Online]. Available: https://www.fuzzingbook.org/html/Reducer.html
[9] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *ESEC/FSE*. ACM, 2021.
[10] M. Zhang, Z. Xu, Y. Tian, X. Cheng, and C. Sun, "Toward a better understanding of probabilistic delta debugging," in *ICSE*, 2025.
[11] P. Hayet, S. H. Tan, and V. Terragni, "CrashJS: A NodeJS benchmark for automated crash reproduction," in *MSR*, 2024.
[12] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *ICSE*, 2006.
[13] R. Hodován and A. Kiss, "Modernizing hierarchical delta debugging," in *A-TEST*, 2016.
[14] A. Kiss, R. Hodován, and T. Gyimóthy, "HDDr: A recursive variant of the hierarchical delta debugging algorithm," in *A-TEST*, 2018.
[15] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *ICSE*, 2018.
[16] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *CCS*, 2018.
[17] R. Hodován and A. Kiss, "Practical improvements to the minimizing delta debugging algorithm," in *SEAA*, 2016.
[18] A. Kiss, "Generalizing the split factor of the minimizing delta debugging algorithm," *IEEE Access*, 2020.
[19] D. Vince, "Iterating the minimizing delta debugging algorithm," in *A-TEST*, 2022.
[20] D. R. MacIver, "Adaptive delta debugging," https://www.drmaciver.com/2017/06/adaptive-delta-debugging/, 2017.