C2D2: Extracting Critical Changes for Real-World Bugs with Dependency-Sensitive Delta Debugging

Xuezhi Song[†]
Fudan University
Shanghai, China
songxuezhi@fudan.edu.cn

Yijian Wu[*][†]
Fudan University
Shanghai, China
wuyijian@fudan.edu.cn

Shuning Liu[†]
Fudan University
Shanghai, China
liushuning@fudan.edu.cn

Bihuan Chen[†]
Fudan University
Shanghai, China
bhchen@fudan.edu.cn

Yun Lin
Shanghai Jiao Tong University
Shanghai, China
lin_yun@sjtu.edu.cn

Xin Peng[†]
Fudan University
Shanghai, China
pengxin@fudan.edu.cn

ABSTRACT

Data-driven techniques are promising for automatically locating and fixing bugs, which can reduce enormous time and effort for developers. However, the effectiveness of these techniques heavily relies on the quality and scale of bug datasets. Despite that emerging approaches to automatic bug dataset construction partially provide a solution for scalability, data quality remains a concern. Specifically, it remains a barrier for humans to isolate the minimal set of bug-inducing or bug-fixing changes, known as *critical changes*. Although delta debugging (DD) techniques are capable of extracting critical changes on benchmark datasets in academia, the efficiency and accuracy are still limited when dealing with real-world bugs, where code change dependencies could be overly complicated.

In this paper, we propose C2D2, a novel delta debugging approach for critical change extraction, which estimates the probabilities of dependencies between code change elements. C2D2 considers the probabilities of dependencies and introduces a matrix-based search mechanism to resolve compilation errors (CE) caused by missing dependencies. It also provides hybrid mechanisms for flexibly selecting code change elements during the DD process. Experiments on Defects4J and a real-world regression bug dataset reveal that C2D2 is significantly more efficient than the traditional DD algorithm *ddmin* with competitive effectiveness, and significantly more effective and more efficient than the state-of-the-art DD algorithm ProbDD. Furthermore, compared to human-isolated critical changes, C2D2 produces the same or better critical change results in 56% cases in Defects4J and 86% cases in the regression dataset, demonstrating its usefulness in automatically extracting critical changes and saving human efforts in constructing large-scale bug datasets with real-world bugs.

[*]Corresponding author
[†]Also affiliated with Shanghai Key Laboratory of Data Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '24, September 16–20, 2024, Vienna, Austria

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652129

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**; **Software testing and debugging**.

KEYWORDS

Delta Debugging, Critical Changes, Probabilistic Model

ACM Reference Format:
Xuezhi Song, Yijian Wu, Shuning Liu, Bihuan Chen, Yun Lin, and Xin Peng. 2024. C2D2: Extracting Critical Changes for Real-World Bugs with Dependency-Sensitive Delta Debugging. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652129

1 INTRODUCTION

Locating and fixing bugs is a time-consuming and labor-intensive task for developers. Data-driven techniques [18, 23, 25, 26] hold promise in efficiently locating and automatically fixing bugs. However, the effectiveness of such techniques heavily relies on the quality and scale of bug datasets. Currently, real-world bug datasets [5, 34, 39, 43] are typically derived from actual commits in code repositories, where some source code changes may be unrelated to the bug, such as feature additions or refactorings. The quality of bug datasets depends on whether the minimum set of changes that cause or fix bugs, also known as *critical changes*, is precisely annotated. Hence, efficiently identifying precise critical changes is valuable for high-quality bug datasets.

Delta debugging (DD) [46] provides an automatic way to identify critical changes for bugs based on testing feedback. The basic idea is to apply or revert different portions of source code changes before running the test code to check whether the program still contains the bug (T) or not (F). A traditional DD implementation is *ddmin* [47], which basically performs a binary search among all elements. Although *ddmin* is capable of finding critical changes, the computational overhead is enormous because numerous possible combinations of change elements must be tested to decide whether the combination contains critical changes. Various techniques [14, 15, 36] have been proposed to improve the efficiency of the original *ddmin*. However, the core algorithm of *ddmin* was not changed.

ProbDD [44] is a new technical attempt different from *ddmin*, which estimates the probabilities of code change elements being

part of critical changes based on the history of test results. It is reported as the state-of-the-art DD algorithm. Guided by the probabilistic model, ProbDD chooses the change elements that are most likely to be in the critical changes. ProbDD is highly efficient. However, it suffers from the assumption that the code change elements are *independent* to each other. Unfortunately, this assumption does not often hold in real-world code commits. The missing dependent change elements cause compilation errors (CE), which ultimately lead ProbDD to include more elements than expected (as will be explained in Section 2.3).

To mitigate the problem of missing dependencies between code change elements, grouping the elements that have dependencies is a possible choice. Existing techniques, such as DDJ [14], can be used to preprocess source code changes so that the grouped code change elements are independent of each other. However, due to the technical limit of program analysis, the reported dependencies are neither sound nor complete, providing less reliable enhancement to ProbDD.

Solving this dependency problem is not trivial. First, the probabilistic model of ProbDD is inherently not designed for dealing with CE because it has only one probability of a code change element "contained in the critical change set". When encountering CE, the model has no choice but to increase the probability of the code change elements in the complement set; otherwise, the reduction process will stall. Second, ProbDD does not have a mechanism to recognize or rectify dependencies between the code change elements, and thus cannot reduce the critical change set correctly in the cases of unrevealed or incorrect dependencies.

Therefore, to achieve an improved balance between the efficiency and accuracy of extracting critical changes of bugs, we seek a novel DD approach that has the ability to (1) accurately and efficiently select appropriate code change elements during DD iterations and (2) not only utilize the dependencies between code change elements but also rectify erroneous or missing ones to improve the accuracy of DD results.

In this work, we propose C2D2 (inspired by the *Star Wars* character R2-D2), a dependency-sensitive DD approach for critical change extraction. It models and tracks the probability of dependencies between code change elements and is able to resolve CE status by adding and removing dependent elements based on the estimation of the dependency probabilities. Specifically, we design a probability model to calculate the possibility of dependencies reported by program analysis and propose a matrix representation to track potential dependencies by filling and updating the probabilities. We develop a matrix-based search mechanism to search for the code change elements that are more likely to resolve CE. Moreover, we employ hybrid strategies, including heuristics considering the code structure and the number of times for which the code change elements are selected, to improve the selection of code change elements in the iterations of DD process.

In evaluation, we first evaluate the effectiveness and efficiency of C2D2 against the traditional DD approach *ddmin* and the state-of-the-art ProbDD[1], using the Defects4J dataset [20] and a regression bug dataset [39]. The results show that C2D2 is 28.74% and 17.13% more efficient in Defects4J and the regression dataset, respectively, with little loss in the number of successful reductions, compared to *ddmin*. Compared to ProbDD, it increases the number of optimal reductions by 37.59% and improves the reduction rate by 14 percentage points (*pts*) in the regression dataset, with 4.38% improvement in efficiency. Second, we conduct an ablation study and observe the contributions of each technical decision of C2D2. Third, we compare the critical changes extracted by C2D2 with human-isolated critical changes. We find that 56% cases in Defects4J and 86% cases in the regression dataset are the same as or better than human-isolated ones. By examining the cases, we confirm that C2D2 is helpful for automatically extracting critical changes when constructing large-scale bug datasets.

In summary, this work makes the following contributions.

- We propose C2D2, a novel dependency-sensitive DD approach to extracting critical changes of real-world bugs, which models and tracks the dependencies between code change elements and has the ability to rectify incorrect dependencies between code change elements.
- We conduct experiments to demonstrate the improvement in effectiveness and efficiency of C2D2 over existing DD techniques on critical change extraction tasks.
- We implement the tool C2D2 and publicize the source code that can be used to build large-scale bug datasets with automatically annotated critical changes.

2 BACKGROUND AND MOTIVATING EXAMPLE

We first introduce the concept of delta debugging and its usage for critical change extraction. Then, we introduce the key idea of probabilistic delta debugging. Finally, we discuss its limitation when dealing with code elements that have unrevealed dependencies through a motivating example.

2.1 Delta Debugging (DD)

DD is an automatic process to find a subset of elements while preserving a certain property [14, 44, 47]. In critical change extraction, the elements are the source code changes that can be counted in terms of chunks, lines, or any specified units. The universe is all subsets of the code changes, which we denote as $\mathbb{X}$. Let $\phi : \mathbb{X} \rightarrow \{T, F\}$ be the function where $\phi(X)$ tests whether the set of code change elements $X \in \mathbb{X}$ contains the critical changes (T) or not (F).

Specifically, for bug-inducing changes, the function $\phi(X)$ is to run the test that is related to the bug in the buggy revision by reverting the code change elements in X. If the test passes, $\phi(X) = T$, meaning that X contains the critical changes that induce the bug. For bug-fixing changes, the function $\phi(X)$ is to run the bug-related test in the bug-fixing revision by reverting the code change elements in X. If the test fails, $\phi(X) = T$, meaning that X contains the critical changes that fix the bug.

Extracting critical changes with DD can be formally formulated as a task to find a smallest set of code change elements (i.e., Eq. 1).

$$X^* = \text{argmin}_{X' \in \mathbb{X}} |X'| \quad s.t. \quad \phi(X^*) = T \tag{1}$$

[1] The code-change-element-dependency detection algorithm provided by DDJ is used in all approaches for a fair comparison.

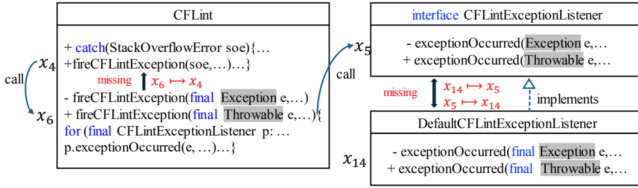

Figure 1: An example of missing code change dependencies

In other words, a DD process is to search for the smallest non-empty set X^* of code changes that contains the critical changes by reducing the running set X'.

2.2 Probabilistic Delta Debugging (ProbDD)

ProbDD proposes a probabilistic model that learns the probability of each change element $x_i \in X$ to be chosen in the running set. The probability of each change element is calculated based on the test results during the DD process. ProbDD selects a running set X' from the set of all elements by iteratively eliminating the element x_j with the lowest probability until the expected gain starts to decrease [44]. ProbDD maintains the probabilities by evaluating $\phi(X')$. If $\phi(X') = T$, the probabilities of all eliminated elements are set to zero; otherwise, ProbDD increases the probabilities of the eliminated elements. Details of the probability-increasing algorithm can be found in the literature [44].

ProbDD distinguishes itself from traditional DD algorithms by not relying on fixed partitions of the interesting element set, and thus is able to flexibly and quickly focus on a small set of the elements that has the highest probability of being the minimal set X^*. However, an important assumption of the probabilistic model is that the elements under consideration are *independent* from each other. If a code change element A depends on another code change element B, the probability of B may be misleadingly increased when the running set X' contains A but does not contain B because the result of $\phi(X')$ will be F due to a compilation error (CE). We present the following motivating example to clarify this argument.

2.3 A Motivating Example

We present a bug-inducing commit[2] to demonstrate how ProbDD iterates to find the critical bug-inducing changes. We use DDJ to group the preliminary code changes into independent change elements by detecting the dependencies between code changes at the granularity of the AST nodes. In this commit, 23 "independent" code change elements, namely $X = \{x_1, x_2, ...x_{23}\}$, are identified. However, due to the limitation of DDJ, three dependencies are missing: $x_6 \mapsto x_4$, $x_{14} \mapsto x_5$, and $x_5 \mapsto x_{14}$, where $a \mapsto b$ denotes "element a depends on element b", which means that if the change a is reverted, b must revert; otherwise CE will occur. Figure 1 details the elements x_4, x_5, x_6, and x_{14}. Code change element x_4 introduces a `catch` block to catch `StackOverflowError` errors and call `fireCFLintException`; x_6 correspondingly changes the type of parameter `e` of `fireCFLintException` from `Exception` to `Throwable` to accept the new type `StackOverflowError`. We say $x_6 \mapsto x_4$ because if we revert x_6 but do not revert x_4, then the program would not compile because of the type incompatibility between `StackOverflowError` and `Exception`. The dependencies between x_5 and x_{14} are missing because DDJ does not properly treat class inheritance and interface implementation.

[2] https://github.com/cflint/CFLint/commit/4731bf45805725bb1d3eac58f9fad8a4b8701f3f

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	
	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	
1	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.24	0.24	0.24	0.24	0.24	0.24	0.15	0.15	0.15	0.15	
2	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.24	0.24	0.24	0.24	0.24	0.24	0.32	0.32	0.32	0.32	
3	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.36	0.36	0.36	0.36	0.24	0.24	0.32	0.32	0.32	0.32	
4	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.36	0.36	0.36	0.36	0.4	0.4	0.52	0.32	0.32	0.32	
5	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.36	0.36	0.36	0.36	0.4	0.4	0.52	0.46	0.46	0.46	
6	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F
	0.61	0.61	0.36	0.36	0.4	0.4	0.52	0.46	0.46	0.46	
7	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.61	0.61	0.61	0.61	0.4	0.4	0.52	0.46	0.46	0.46	
8	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.61	0.61	0.61	0.61	0.62	0.62	0.52	0.46	0.46	0.46	
9	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	T
	0.61	0.61	0.61	0.61	0.62	0.62	0.52	0	0	0.46	
10	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	0.61	0.61	0.61	0.61	0.62	0.62	0.52	0	0	1	
11	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	T
	0.61	0.61	0.61	0.61	0.62	0.62	0	0	0	1	
12	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F
	1	0.61	0.61	0.61	0.62	0.62	0	0	0	1	
13	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F
	1	1	0.61	0.61	0.62	0.62	0	0	0	1	
14	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F
	1	1	1	0.61	0.62	0.62	0	0	0	1	
15	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	1	1	1	1	0.62	0.62	0	0	0	1	
16	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	F#
	1	1	1	1	1	0.62	0	0	0	1	
17	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{14}	T
	1	1	1	1	1	0	0	0	0	1	

Figure 2: An example illustrating the iterations of ProbDD

Figure 2 demonstrates part of the iteration process performed by ProbDD[3]. Each iteration is numbered on the left side; the elements in the running set X' are marked in blue; the value below each element is the probability calculated by ProbDD after each iteration; the results of $\phi(X')$ are marked on the right side. Recalling that ProbDD does not recognize compilation errors (CE), we add a hash mark (#) after the F if a CE occurs in that iteration. ProbDD first initializes the probabilities of all elements to the same value 0.15. Then it starts with a random selection of change elements as the running set X' ($\{x_7, x_8, x_9, x_{14}\}$, marked dark green) in Iteration 1. Since X' does not include x_5, dependency $x_{14} \mapsto x_5$ is broken so the program does not compile. However, ProbDD does not consider the CE status but simply evaluates $\phi(X')$ to F, and increases the probabilities of x_1 to x_6 from 0.15 to 0.24. Subsequently, based on the probabilities, ProbDD employs a greedy strategy to select x_1 to x_6 to be tested by ϕ in Iteration 2. Although X' now contains

[3] For ease of understanding, we opt to omit some code change elements that are unrelated to the missing dependencies.

all elements in X^*, $\phi(X')$ is still evaluated as $F\#$ because x_{14} is not in X' and breaks $x_5 \mapsto x_{14}$. This misleads ProbDD to increase the probabilities of the other elements from 0.15 to 0.32.

The similar thing happens in Iterations 3-5 and 7-8, where ProbDD disregards CEs but only increases the probabilities of the change elements.

In Iteration 9, the running set again contains all elements in X^* (which we know later are $\{x_1, x_2, x_3\}$) and the compile succeeds, so $\phi(X') = T$. Then ProbDD sets the probability of x_8 and x_9 to zero to exclude them from the running set. Now that the probabilities of all elements exceed 0.5, ProbDD starts to remove only one element in each iteration. Once $\phi(X') = F$ (including CEs), the probability of the removed element is updated to 1, causing the element to be included in the result. We find that, in Iterations 10, 15, and 16, x_{14}, x_4 and x_5 are erroneously included in the result set, which results in a suboptimal critical change set $\{x_1, x_2, x_3, x_4, x_5, x_{14}\}$.

In this example, we observe an ineffective DD process, yielding a critical change set significantly larger than X^* due to disregarding change element dependencies and the subsequent erroneous test results. In several iterations, the running set already contains X^* but due to CEs, ProbDD is not able to correctly predict which elements are in the critical change set. If CE could be resolved, certain non-critical change elements would have been accurately excluded from running set. For example, adding x_{14} or removing x_5 in Iteration 2 could be a viable attempt.

Algorithm 1: Extracting Critical Changes with C2D2

```
    Input  : V, the target revision; X, the code change elements
             between V and V − 1
    Output : X*, the critical changes
1   M = initialize_matrix(X,V)
2   X' = sample_init_test_set(X)
3   repeat
4       test_result = predict(X')
5       if test_result == NULL then
6           is_CE = compile_and_update_matrix(V, X', M)
7           if is_CE == True then
8               X'_f = resolve_CE(V, X', X, M)
9               test_result = τ(V, X'_f)
10          else
11              test_result = τ(V, X')
12          X' = (|X'_f| < |X'| ? X'_f : X')
13      greedy_search(X', test_result)
14      if test_result is T then
15          X = X'
16          X* = X'
17          reduce_matrix(M, X')
        // select a test set for the next iteration
18      X' = sample(X)
19  until ProbDD.done() is True;
20  return X*
```

3 THE DEPENDENCY-SENSITIVE APPROACH

3.1 Overview

The key idea of our approach is to model and keep track of the dependencies between code change elements so that when CE is encountered during DD process, the dependencies could be used to add or remove code change elements so that CE could be resolved. Moreover, the selection of code change elements during the DD process is also specifically designed to maximize the possibility of achieving the minimal critical change set.

Algorithm 1 presents an overview of our approach. The inputs include a target revision V and a set X containing the code change elements between revision V and its previous revision $V - 1$. The output is the set of critical changes X^* identified from X.

It first initializes the dependency matrix M with the dependencies between any two elements in X (Line 1; see Section 3.2.1). Then, it samples a number of elements from X as the initial running set X' (Line 2; see Section 3.4.1). Next, it iterates to find the smallest subset of X' that includes the critical changes by applying a greedy-search based strategy with consideration of the probabilities of each element being part of the critical changes. The iterations continue until the termination criterion is satisfied (i.e., the probabilities of all elements are either zero or one, or the time budget is reached).

Each iteration starts with predicting whether the testing result $\phi(X')$ is F (Line 4). According to the monotony property [44, 46], which has been widely accepted in the literature, we predict $\phi(X') = F$ if there is a super set of X' whose testing result is F.[4] Otherwise, the prediction result is set to $NULL$, leading to a compilation attempt after reverting the code change elements in X' and updating the matrix accordingly (Line 6; Section 3.2.2). If compilation fails (i.e., CE occurs), it attempts to resolve CE (Line 8; Section 3.3) by finding a new set of elements X'_f, and runs the test with X'_f (Line 9); otherwise, it runs the test with the original X' (Line 11). If X'_f is generated by removing some elements from X', then X' is reduced to X'_f (Line 12). The probability of each element in X' is updated by the greedy-search based algorithm (similar to the probability-update strategy of ProbDD; Line 13) before the dimension reduction of M if $\phi(X') = T$ (Line 14–17; Section 3.2.3). The next iteration begins with a hybrid mechanism for element selection (Line 18; Sections 3.4.2 and 3.4.3).

3.2 Maintaining the Dependency Matrix

3.2.1 Initializing the Matrix. The dependency matrix M is initialized based on the dependencies between code change elements reported by program analysis techniques proposed in DDJ [14]. Given the set X of code change elements between two revisions, we denote the set of dependencies reported by DDJ as $D = \{(x_i, x_j) \mid x_i, x_j \in X \wedge x_i \mapsto x_j\}$.

Since program analysis may produce incorrect dependencies, we assign a probability of δ that a reported dependency is true. If a dependency from one element to another is not reported, we assume that there is still a probability of $1 - \delta$ that the dependency exists. Thus, the dependency matrix $M \in \mathbb{R}^{|X| \times |X|}$ is initialized by Eq. 2.

$$M[x_i][x_j] = \begin{cases} 1 - \delta, & \text{if } (x_i, x_j) \notin D \\ \delta, & \text{if } (x_i, x_j) \in D \end{cases} \tag{2}$$

[4]Strictly speaking, if there is a super set of X' whose testing result is F, we can only predict $\phi(X')$ must not be T but could be F or an *unresolved* status. Here, since we will test the compilability later, assuming $\phi(X') = F$ is acceptable.

where x_i and x_j are elements in X. In our implementation, we set δ as a hyper-parameter smaller than 1, which provides us with the opportunity to identify and rectify erroneous dependencies.

3.2.2 Updating the Matrix. The probabilities of dependencies are updated in the matrix M based on the compilation results.

If compilation fails with running set X' (i.e., $\phi(X') = CE$), it indicates that some element(s) in X' depend on element(s) in the complement $\overline{X'}$. Therefore, the corresponding probabilities of dependency should be increased. We use θ_{ij}, a Bernoulli random variable, to denote whether or not a dependency actually exists from element x_i to element x_j. Furthermore, we assume that each θ_{ij} constitutes mutually independent events, implying that the pairwise dependencies between elements in X are not influenced by other dependencies. Thus, for any $x_i \in X', x_j \in \overline{X'}$, we calculate the probability of x_i depending on x_j using the Bayesian formula in Eq. 3.

$$M[x_i][x_j] = P(\theta_{ij} = 1 | \phi(X') = CE) = \frac{P(\theta_{ij} = 1) * P(\phi(X') = CE | \theta_{ij} = 1)}{P(\phi(X') = CE)} \tag{3}$$

Here, we have $P(\phi(X') = CE | \theta_{ij} = 1) = 1$ because a CE must occur if x_i depends on x_j. $P(\theta_{ij} = 1)$ is the current value of $M[x_i][x_j]$. $P(\phi(X') = CE)$ is the probability that there exists at least one dependency from X' to $\overline{X'}$, which is $1 - \prod_{\forall x_a \in X', x_b \in \overline{X'}} (1 - M[x_a][x_b])$. Therefore, we update M by Eq. 4 when compilation fails.

$$M[x_i][x_j] = \frac{M[x_i][x_j]}{1 - \prod_{\forall x_a \in X', x_b \in \overline{X'}} (1 - M[x_a][x_b])} \tag{4}$$

If compilation succeeds, we update M by Eq. 5.

$$M[X'][\overline{X'}] = 0 \tag{5}$$

It means that all dependencies from X' to $\overline{X'}$ are confirmed as 0 such that erroneous relationships in the matrix are removed[5].

3.2.3 Reducing the Dimension of the Matrix. When the testing result is T (i.e., $\phi(X') = T$), it means that $\overline{X'}$ does not contain any critical changes. This allows us to reduce the dimension of M by removing $M[\overline{X'}][\overline{X'}]$ from the matrix by Eq. 6.

$$M = M[X'][X'] \tag{6}$$

3.3 Resolving Compilation Errors

We transform the task of resolving CE for the running set X' into a search-based program repair problem constrained within a finite space. Specifically, we employ a matrix-based search (MBS), which uses the dependency probability matrix to guide the process of generating CE-resolving sets.

3.3.1 Matrix-based Search (MBS). Given a running set X' satisfying $\phi(X') = CE$, MBS iteratively attempts to generate CE-resolving sets by incorporating elements from $\overline{X'}$ or removing elements from X'. Algorithm 2 presents the process of MBS, which makes a maximum number of K attempts to generate CE-resolving sets. In each attempt, MBS searches for an add-element CE-resolving set (Line 21) and a remove-element CE-resolving set (Line 24). All CE-resolving sets are stored in a list C. If a newly-generated set already exists in the list, it is discarded. The stochastic element searching process is guided by the dependency probabilities between the elements in X' and $\overline{X'}$ maintained in the matrix.

Algorithm 2: Matrix-based Search

Input : M, the dependency matrix; X', the current set of elements; K, the maximum number of attempts
Output: C, the list of CE-resolving sets

```
1  function select_add_stochastic ()
2      var E_add = [], X_consider = X'
3      while |X_consider| > 0 do
4          var M_slice = M[X_consider][X̄' − E_add]
5          var w^{1×|X̄'−E_add|} = column_max(M_slice)
6          var X_sel = select_stochastic(X̄' − E_add, w)
7          E_add.append(X_sel)
8          X_consider = X_sel
9      return E_add
10 function select_remove_stochastic ()
11     var E_remove = [], X_consider = X';
12     while |X_consider| > 0 do
13         var M_slice = M[X_consider][X̄' − E_remove]
14         var v^{1×|X_consider|} = (row_max(M_slice))^T
15         var X_sel = select_stochastic(X_consider, v)
16         E_remove.append(X_sel)
17         X_consider = X_consider − X_sel
18     return E_remove
19 C = ∅
20 for i in range(0, K) do
21     E_add = select_add_stochastic ()
22     if C does not contain X' ∪ E_add then
23         C.add(X' ∪ E_add)
24     E_remove = select_remove_stochastic ()
25     if C does not contain X' − E_remove then
26         C.add(X' − E_remove)
27 return C
```

3.3.2 Adding Elements to X'. MBS searches for elements from $\overline{X'}$ with a stochastic strategy based on the values in the dependency matrix. First, the submatrix $M_{slice} = M[X'][\overline{X'}]$ which contains the probabilities of dependencies from the elements in X' to those in $\overline{X'}$ is considered. MBS constructs a vector $w \in \mathbb{R}^{|\overline{X'}|}$ by concatenating the maximum values for each *column* in M_{slice} (Line 5). Therefore, each value in the vector w corresponds to the maximum probability of an element in $\overline{X'}$ depended on by some elements in X'.

Then, MBS uses a stochastic strategy to select elements from $\overline{X'}$ (Line 6). Whether or not an element in $\overline{X'}$ is selected is based on the corresponding probability value represented in w and is independent of other elements. The higher probability an element is depended on by elements in X', the more likely it is selected to be added to X'. Since it is a stochastic selection strategy, it is possible that no elements are selected. If this occurs, the current iteration is

[5] Here, we use $M[A][B]$ to denote a submatrix of M, whose rows are identified by the set of elements A and columns are identified by the set of elements B. The form $M[A][B]$ is similar to $M[x_i][x_j]$. The only difference in semantics is that $M[x_i][x_j]$ denotes a value identified by elements x_i and x_j whereas $M[A][B]$ denotes a submatrix.

terminated. The selected elements in each iteration are denoted as X_{sel} and are added to a temporary set E_{add} (Line 7).

Note that a selected element in X_{sel} to be added to X' may also depend on other elements in $\overline{X'}$. Therefore, MBS cascadely considers which elements in $\overline{X'}$ are depended on by elements in X_{sel} (Line 8). A new submatrix $M[X_{sel}][\overline{X'} - X_{sel}]$ is considered to select more elements from $\overline{X'} - X_{sel}$ into E_{add}. This process continues until no more element is selected. A new CE-resolving set is generated with the union of X' and all elements in E_{add} (Line 23).

3.3.3 Removing Elements from X'. Similar to the process in Section 3.3.2, MBS start removing elements from X' by considering M_{slice}. The difference is that a vector $v \in \mathbb{R}^{|X'|}$ is constructed by concatenating the maximum values for each *row* in M_{slice} (Line 14). Hence, each value in the vector v corresponds to the maximum probability of an element in X' depending on some elements in $\overline{X'}$.

Then, MBS applies the same strategy (Line 15) as in the process in Section 3.3.2 to select elements from X' into the temporary set E_{remove} (Line 16). Subsequently, the rest of the elements in X' (Line 17) are considered to be removed based on the probability in which they depend on the previously selected elements. This process iterates until no element is selected. A new CE-resolving set is generated by removing all elements in E_{remove} from X' (Line 26).

3.3.4 Discussions on the Maximum Number of Attempts K. MBS returns a list C of all sets of elements that are potentially able to resolve CE. The size of C is not greater than $2 \times K$. We use a dynamic value of K that increases with the count of CEs that are encountered during the original ProbDD process. Empirically, we set $K = i \times \ln(|X|)$, where i denotes the number of ProbDD iterations and $|X|$ denotes the number of elements currently under consideration ($X = X' \cup \overline{X'}$). We opt to apply this strategy because in the early stage of ProbDD, if the number of elements is huge, resolving a CE could be extremely expensive. Therefore, we spend less effort in resolving the CE in the early stage and increase the effort when more CEs are encountered. Since the complexity of ProbDD is theoretically $O(N)$ where N is the number of elements, the order of magnitude of i is also $O(N)$. Therefore, the maximum attempt number K is on the $O(N\ln(N))$ order of magnitude.

3.4 Selecting Elements for the Running Set

During DD iterations, we apply three strategies to choose appropriate code change elements as candidates for the critical change.

3.4.1 "Cold-Start": Selecting Elements for the First Iteration. When the DD process starts, we try to select the code change elements which, if reverted, are more likely to (1) affect program behavior and (2) pass compilation. To achieve this, we opt *not* to select the elements that are *inserted* declarations of classes or class members (including fields and methods). The underlying rationale is that the new declarations are typically used by other code changes. If they are reverted independently, the program is very likely to fail compilation. Hence, excluding them from the first iteration is a cheap decision.

3.4.2 "Equal-Chances": Hybrid Element Selection Mechanism during Iteration. The DD process of C2D2 basically follows a greedy search strategy [44], similar to ProbDD, to select elements for a new iteration based on the probability of being part of the critical change. However, the greedy strategy overlooks potential errors in the probability calculation. Therefore, we introduce a new random selection strategy, which is used with probability ξ as a substitute for the original ProbDD mechanism. With the new strategy, elements are selected based on the *selection frequency*. Specifically, we use $T = \{t_1, t_2, ...t_n\}$ to record the number of times each x_i in X is selected. Next, we calculate the selection frequency $f_i = t_i/(max(T) + min(T))$ for x_i. Then, we use $1 - f_i$ to denote the weight for each element to be selected. Therefore, elements with a lower selection frequency will have more chances of being selected. This strategy enables C2D2 to find new solutions by breaking the constraints set by ProbDD.

3.4.3 "Start-Afresh": Searching for a New Set. When a CE-resolving attempt fails, we need a new subset of the search space X to continue the DD process. To search for it, we dynamically select elements with lower probabilities of depending on other elements in X.

Given a current search space X, we retrieve the maximum value of each row in M to construct a maximum dependency vector $v \in \mathbb{R}^{|X|}$. This vector v represents the maximum probability of dependency of $x_i \in X$ on the other elements in X. Next, we employ a dynamic threshold η, ranging from $[0, \max(v))$, to incrementally select elements whose value in v is smaller than η. When the selected set of elements fails to compile, we increase the value of η by $\frac{\max(v)}{K}$. Here, K continues to represent the maximum number of search attempts. Sets of elements that have been previously tried for compilation or testing are discarded. Thus, the method starts with elements that have no dependencies, dynamically selects new combinations of elements based on matrix variations, and searches for a compilable set within the constraint of K attempts to continue the DD process.

3.5 Revisiting the Motivating Example

In this section, we demonstrate how C2D2 accurately locates the critical bug-inducing changes of the motivating example. Key iterations of the DD process are illustrated in Figure 3. Starting from the 23 code change elements produced by DDJ, C2D2 first starts the *cold-start* component and excludes five elements in the first step. The excluded elements are mainly *import* declarations and implementation of new features. Next, C2D2 uses the greedy-search strategy to select X' in Iteration 2. The test result is CE. Then, C2D2 applies MBS to resolve CE and succeeded by adding six elements from $\overline{X'}$. The test result is T, so the rest of the elements are permanently excluded from the DD process.

Iterations continue until the 12th one, when the *equal-chances* component is triggered, leading to the selection of $\{x_2, x_{13}\}$ based on the frequency of element selection. The compilation succeed and the test result is F. Prior to this iteration, DDJ provided the dependency $x2 \mapsto x3$. As the compilation with $\{x_2, x_{13}\}$ does not fail, we confirm that the dependency $x2 \mapsto x3$ is invalid and the dependency matrix is updated by setting $M[x2][x3]$ to 0. In this iteration, C2D2 exhibits the ability to rectify erroneous dependencies.

In Iteration 19, C2D2 selects elements $\{x_1, x_2, x_3\}$, which is identical to X^*. The test result is true. C2D2 continues to select a subset $\{x_1, x_2\}$ in Iteration 20, where a CE occurs. Then, our approach tries to add x_3 or remove x_1 but all attempts have been previously tested. So C2D2 triggers another *start-afresh* and selects a new

	Components	Operations	Results
	DDJ	provide X = {x_1, x_2, …, x_{23}}	
1	Cold-Start	exclude {x_8, x_9, x_{11}, x_{12}, x_{17}}	T
2	Greedy-Strategy	select{x_1,x_2,x_{10},x_{13},x_{14},x_{18},x_{19},x_{20},x_{21}}	CE
	MBS	add {x_3, x_4, x_5, x_6, x_{16}, x_{22}}	T
		……	
12	Equal-Chances	select {x_2, x_{13}}	F
13	Greedy-Strategy	select {x_1, x_3, x_5, x_{13}, x_{15}, x_{16}, x_{18}}	CE
	MBS	add {x_{14}, x_{20}}	F
14	Greedy-Strategy	select {x_1, x_2, x_3, x_6, x_{14}, x_{18}, x_{20}, x_{22}}	CE
	MBS	remove {x_6, x_{14}, x_{18}, x_{20}}	T
		……	
19	Greedy Strategy	select {x_1, x_2, x_3}	T
20	Greedy Strategy	select {x_1, x_2}	CE
	MBS	add {x_3}, remove {x_1}	
	Start-Afresh	select {x_2, x_3}, retain {x_1, x_2}	F#
	C2D2	**Done & output {x_1, x_2, x_3}**	

Figure 3: C2D2 bug-inducing reduction process

combination of elements $\{x_2, x_3\}$. However, this change set has also been tested and removing or adding elements does not create new change sets. Therefore, C2D2 retains the selection $\{x_1, x_2\}$ and updates the test result to *F#*. Finally, the termination criteria for the process are met and the DD process is completed.

In this case, C2D2 required only 20 iterations and 12 additional compilation attempts to accurately identify the critical changes as $\{x_1, x_2, x_3\}$, which is better than the result produced by ProbDD.

4 EVALUATION

We evaluate C2D2 by answering three research questions.

RQ1 Effectiveness and Efficiency: What is the effectiveness and efficiency of C2D2 compared to existing DD algorithms in the task of extracting critical changes for bugs?

RQ2 Ablation Study: What are the contributions of each technical decision, including cold-start, equal-chances, and start-afresh, to the effectiveness and efficiency of C2D2?

RQ3 Data Quality and Usefulness: What is the quality of the critical changes extracted by C2D2 compared to those isolated by humans? Is C2D2 useful for constructing large-scale bug datasets?

4.1 Evaluation Setup

4.1.1 Dataset. DD algorithms are capable of extracting critical changes for both *bug fixes* and *bug induces*.

To evaluate the performance of C2D2 on *bug-fixing* change reduction tasks, we use Defects4J [20], a manually-verified bug dataset widely used in the literature. It consists of 835 bugs and their fixes collected from real-world Java applications. For each bug, it provides the *original* buggy revision (V_{obug}) and bug-fixing revision (V_{ofix}), between which the code differences contain significant bug-irrelevant changes [19]. We use the code differences between V_{obug} and V_{ofix} as input for the DD algorithms. Defects4J also provides critical changes of bug fixes, which are manually minimized and verified by humans. These human-isolated critical changes are used as the ground-truth. During our evaluation, we found 26 bugs whose revisions V_{obug} and V_{ofix} were not accessible; therefore, 809 bugs are finally used in our evaluation.

Table 1: Statistics of Code Change Elements of the Bugs in the Regression Dataset and Defects4J

Number of Elements	Regression Dataset	Defects4J
Minimum	1	1
25th Percentile	7	2
Median	17	3
75th Percentile	77	7
Maximum	2,136	108

For *bug-inducing* change reduction tasks, we use the regression bug dataset constructed by RegMiner [39]. It contains 1,035 regression bugs and, to the best of our knowledge, is the only large-scale dataset that contains bug-inducing changes, even though the changes have not been minimized by human inspection. To ensure the quality of the dataset, we manually examined the data and identified 10 entries that were either duplicates or related to flaky tests. Therefore, 1,025 regression bugs are used in our evaluation. We also observe that the number of code change hunks in the regression dataset is significantly higher than that of Defects4J, as shown in Table 1. Therefore, we can compare the performance of the considered DD algorithms when they are used to extract critical changes on datasets with different complexity.

4.1.2 RQ1 Setup. To address **RQ1**, we compared C2D2 with representative DD algorithms *ddmin* [47] and ProbDD, on both tasks of extracting *bug-fixing* and *bug-inducing* critical changes on the corresponding datasets. The DDJ [14] dependency detection mechanism was integrated in all DD algorithms for a fair comparison. DDJ employs Diff/TS [13], a widely-used code differencing tool, to extract code changes at the granularity of AST nodes. It uses syntax- and semantics-aware code change dependency construction techniques to group related code change elements, as this technique has been shown to be an effective optimization for DD algorithms. For simplicity of presentation, we directly use *ddmin* and ProbDD as the names of baselines. The metrics of the efficiency and effectiveness of each algorithm are described in Section 4.1.5.

4.1.3 RQ2 Setup. To address **RQ2**, we conducted an ablation study with both Defects4J and the regression dataset. We independently disabled *Cold-Start*, *Equal-Chances*, and *Start-Afresh*, one at a time, to investigate the contributions of each individual component. These versions are denoted as C2D2$_{\neg CS}$, C2D2$_{\neg EC}$, and C2D2$_{\neg SA}$, respectively. The disabled components were replaced by the original element selection strategies of ProbDD. Then we observed how effectiveness and efficiency changed in these ablated versions. We also turn off all three components to see how C2D2 performs with the dependency-matrix only.

4.1.4 RQ3 Setup. To address **RQ3**, critical changes isolated by humans need to be identified. For Defects4J, we directly used the critical bug-fixing changes that had already been minimized and verified by humans. For the regression dataset, 65 bugs were randomly selected by two authors, who are graduate students majored in software engineering with at least three years of Java development experience, independently annotated critical changes by manually analyzing the changes in the source code. The inter-rate agreement

measured by Cohen's Kappa was 0.78, which exhibited validity of the annotations. For those inconsistently annotated cases, the two graduate students and an additional doctoral candidate student (also an author majored in software engineering and with rich experience in Java development) discussed the results until an agreement was reached. For each bug, the critical changes produced by C2D2 are compared to the human-isolated critical changes.

Additionally, BugBuilder[19], the state-of-the-art bug dataset construction tool that is not based on DD algorithms, is capable of identifying critical changes for bug fixing. Therefore, we also performed comparisons with the critical changes extracted by BugBuilder.

4.1.5 Metrics. The following metrics are typically used in previous work on DD [14, 15, 44, 47].

The effectiveness is measured primarily by the following metrics.

- Number of Successful Reductions (#Success): the number of bugs for which DD successfully terminates (i.e., not exceeding a 2-hour limit) and produces an outcome.
- Average Reduction Rate (Reduction%): $\frac{|X|-|X^*|}{|X|}$, where X^* is the set of critical changes produced by the algorithm being evaluated, while X is the original set of code change elements.
- Number of Optimal Results (#Optimal): the number of critical change sets that are the smallest among the outcomes of all algorithms. If the critical change sets for the same bug are of the same size, they are all considered optimal.

The efficiency is measured primarily by the following metrics.

- Average Time Consumed per Bug (Time): time in seconds that the algorithm takes on average for a successful reduction on a bug. Timeout cases are excluded.
- Average Count of Test Runs per Bug (#Test): the number of tests that have been run on each successful reduction, on average.
- Average Count of Additional Compilations per Bug (#Ad.Compile): Only for C2D2 since C2D2 attempts to resolve the CE status. This measure does not apply to other algorithms.

The measures that we adopt for comparing the outcomes of C2D2 with the human-isolated changes include the following metrics.

- Number of the Same Outcomes (#Same): The critical change sets that are the same as human-isolated ones are counted.
- Number of Better Outcomes (#Better): The critical change sets that are *proper subsets* of human-isolated ones are counted.
- Number of Worse Outcomes (#Worse): The critical change sets that are super sets of human-isolated ones are counted. If a tool terminates with an error or timeout when reducing the changes of a specific bug, it is also counted as Worse.
- Number of Different Outcomes (#Diff): The critical change sets that are neither the same as nor better/worse than human-isolated ones are counted, even if the sets of critical changes are smaller than the human-isolated change.

The critical changes produced by BugBuilder on Defects4J are also compared to human-isolated changes against the above metrics.

4.1.6 Implementation. We implemented C2D2 on the architecture of DDJ [14] by incorporating the dependency matrix-based algorithm into the DD.py [45], which was initially created by Zeller. DDJ extracts critical bug-fixing changes in Java. It incorporates the Diff/TS framework and integrates 500 rules specifying dependencies between code change elements. DDJ employs hierarchical delta debugging (HDD). For a fair comparison, we reimplemented ProbDD by updating DD.py in DDJ with the ProbDD implementation provided by the authors of ProbDD so that implementation differences other than the core algorithms were eliminated. We also decoupled `_compile` from the `_test` function such that we are able to carry out compilation and tests seperately. Our evaluation was conducted on an Ubuntu 20.04 server with a 16-core 32-thread Silver 4208 (2.10GHz) CPU and 64 gibibyte/GiB of RAM.

4.2 RQ1: Effectiveness and Efficiency Evaluation

We present the performance of C2D2 on bug-fixing reduction tasks with Defects4J and on bug-inducing reduction tasks with the regression dataset.

Bug-fixing Reduction on Defects4J. As shown in the upper three rows of Table 2, C2D2 outperforms both *ddmin* and ProbDD. For effectiveness, C2D2 successfully produced critical bug-fixing changes on 702 bugs (out of 809), 6.69% more than both *ddmin* and ProbDD, both of which only suceeded in 658 bugs. Among successful reductions, C2D2 produced 696 optimal results, surpassing *ddmin* and ProbDD by 6.75% and 23.62%. C2D2 achieved the highest average reduction rate of 38.23%, 0.23 percentage points (*pts*) higher than *ddmin* and 5.32 *pts* higher than ProbDD.

For efficiency, we consider the 580 bugs that *all* three DD algorithms successfully produce an outcome, for fair comparison. C2D2 is the fastest in terms of the average time consumed per bug, 28.74% and 26.38% faster than *ddmin* and ProbDD, respectively, even if it required 6.56 additional compilations.

Bug-inducing Reduction on Regression Bugs. As shown in the upper three rows of Table 3, C2D2 also outperforms *ddmin* and ProbDD on multiple metrics. For effectiveness, C2D2 produced 571 optimal results, outperforming *ddmin* and ProbDD by 1.60% and 37.59%, respectively. C2D2 reached the highest average reduction rate of 45.06%, which is 5.03 *pts* higher than *ddmin* and 14.44 *pts* higher than ProbDD. However, C2D2 succeeded in 605 bugs out of 1,025, slightly less than *ddmin* and ProbDD by 15 (2.42% lower) and 59 (8.89% lower), respectively.

Among all successful results produced by *ddmin* and ProbDD, we found 66 and 77 cases, respectively, in which C2D2 failed to produce critical changes. In 41 out of the 66 cases completed by *ddmin* and all 77 cases by ProbDD, the critical change sets are exactly the same as the input set of code change elements, meaning that the reduction rate is 0% ($X^* = X$). Moreover, in all of these cases, the code change elements in the input set have complicated interrelationships, which may cause C2D2 to continuously explore the dependencies before the time limit is reached.

For efficiency, we consider the 488 bugs that *all* three DD algorithms successfully produce an outcome, for fair comparison. C2D2 takes 987.91 seconds to extract critical changes for a bug on average, which is 17.13% faster than *ddmin* and 4.38% faster than ProbDD. For each critical change extraction task, C2D2 only need to run 16.11 tests on average, which is only 37.08% of *ddmin* and 47.00% of ProbDD, which means less test resource consumption. Even if C2D2 had to attempt averagely 21.87 additional compilations for each task, the sum of the number of compilations and the number

Table 2: Effectiveness and Efficiency on Defects4J (809 bugs in total)

	Effectiveness			Efficiency (580 bugs succeeded in common)		
	#Success	#Optimal	Reduction%	Time (sec.)	#Test	#Ad.Compile
ddmin	658	652	38.00%	958.02	6.94	\
ProbDD	658	563	32.91%	927.23	7.35	\
C2D2	**702**	**696**	**38.23%**	**682.66**	4.58	6.56
C2D2$_{\neg CS}$	606 (-13.68%)	594 (-14.66%)	37.75%	821.44	6.34	9.90
C2D2$_{\neg EC}$	671 (-4.42%)	659 (-5.32%)	37.92%	770.70	4.35	7.85
C2D2$_{\neg SA}$	641 (-8.69%)	621 (-10.78%)	37.16%	790.85	5.23	11.45

Table 3: Effectiveness and Efficiency on Regression Bugs (1025 bugs in total)

	Effectiveness			Efficiency (488 bugs succeeded in common)		
	#Success	#Optimal	Reduction%	Time (sec.)	#Test	#Ad.Compile
ddmin	620	562	40.03%	1,192.07	43.45	\
ProbDD	**664**	415	30.62%	1,033.12	34.28	\
C2D2	605	**571**	**45.06%**	**987.91**	16.11	21.87
C2D2$_{\neg CS}$	525 (-13.22%)	452 (-20.84%)	38.38%	1,135.62	20.68	41.86
C2D2$_{\neg EC}$	529 (-12.56%)	450 (-21.19%)	41.70%	1,044.57	15.75	34.44
C2D2$_{\neg SA}$	533 (-11.90%)	460 (-19.44%)	42.43%	1,077.18	17.95	38.07

of tests is still slightly less than the number of tests of *ddmin* and similar to that of ProbDD.

In summary, C2D2 is significantly more effective than ProbDD by increasing the number of optimal reductions by 23.62% in Defects4J and 37.59% in the regression dataset. The reduction rates in the two datasets are improved by 5.32 and 14.44 *pts*, respectively. It is also 26.38% and 4.38% faster than ProbDD in the two datasets, respectively. Compared to *ddmin*, C2D2 achieved 28.74% and 17.13% efficiency improvement in Defects4J and the regression dataset, respectively, with a comparable number of successful reductions, a slightly-increased number of optimal reductions, and a slightly-improved reduction rate.

Answer to RQ1: C2D2 outperforms ProbDD in both Defects4J and the regression dataset. It achieves significantly higher efficiency than *ddmin*, with competitive effectiveness in Defects4J and higher effectiveness in the regression dataset.

4.3 RQ2: Ablation Study

The results of our ablation study on Defects4J and the regression dataset are reported in the lower three rows of Table 2 and Table 3, respectively.

Cold-Start. When *Cold-Start* is disabled, there is a significant decrease in both the effectiveness and efficiency of C2D2 on all measures in both datasets. We observe large decrease of optimal results and the reduction rate. This indicates that the Cold-Start component imposes a great contribution to minimize running set. To further validate this, we conducted an in-depth analysis of the reduction process of C2D2. Specifically, we found that, in the first iteration, C2D2 yielded a T result for 360 regression bugs, with an average reduction rate of 26.54%, and for 301 bugs in Defects4J, with an average reduction rate of 42.78%, which produced a significantly better start of the DD process. We also observe a significant decrease in efficiency when cold-start is disabled, confirming the usefulness of cold-start in shortening the DD process.

Equal-Chances and Start-Afresh. When *Equal-Chances* or *Start-Afresh* is disabled, both #Success and #Optimal decrease severely in both datasets, showing positive contributions of these two components. Meanwhile, we observe slight drop of Reduction%. Specifically, the drop of Reduction% in Defects4J is much smaller than in the regression dataset. Since the code change set of regression bugs are on average larger than those in Defects4J, we conclude that these components make more contributions to complicated reduction tasks than to simpler ones.

We also observe a remarkable decrease in efficiency in both datasets when either of components is disabled, which shows that the introduction of these additional mechanisms for selecting code change elements has a positive contribution to the DD process. Specifically, disabling these components results in less flexibility in element selection, and thus causes more additional compilations. Therefore, we conclude that they are useful mechanisms in the DD process when selecting code change elements during the iterations. Additionally, we also turned off all three components in order to understand the improvements brought by combination of these components. We find that, with all three components turned off, the reduction rates of C2D2 on regression bugs and defects4j decrease to 37.80% (-7.26 *pts*) and 37.05% (-1.18 *pts*), respectively, but are still 7.18 *pts* and 4.14 *pts* higher than those of ProbDD, respectively. This demonstrates the superiority of the hybrid strategy and also indicates that the MBS component effectively mitigated the impact of CEs on ProbDD.

Answer to RQ2: *Cold-Start*, *Equal-Chances*, and *Start-Afresh* make positive contributions for better effectiveness and higher efficiency.

4.4 RQ3: Data Quality and Usefulness

Table 4 presents the number of reduction results produced by C2D2 that are the same as, better/worse than, and different from the

critical changes isolated by humans. We also include the statistic data of BugBuilder [19] in this table as a reference.

Table 4: Comparing the Critical Changes Extracted by C2D2 with Human-Isolated Changes

Dataset	Tool	#Same	#Better	#Diff	#Worse
Defects4J (809 bugs)	C2D2	210 (25.96%)	249 (30.78%)	33 (4.08%)	317 (39.18%)
	BugBuilder	308 (38.07%)	12 (1.48%)	0 (0%)	489 (60.45%)
Regression (65 bugs)	C2D2	44 (67.69%)	12 (18.46%)	2 (3.08%)	7 (10.77%)

First, we find that, compared to BugBuilder which reports reduction results only on Defects4J dataset, C2D2 produces significantly more *Better* cases (249 of C2D2 vs. 12 of BugBuilder) and less *Worse* cases (317 vs. 489).

Second, we examine the 35 *Different* cases (33 in Defects4J and 2 in the regression dataset) and find that the reduction results in 34 cases are also correct critical changes because there are multiple ways to fix the bugs. Only one erroneous reduction result is identified in the regression dataset. The incorrect case comes from the project *fastjson* at commit `6e53cca`, where a flaky test exists. As shown in Listing 1, the test code is designed to validate the function of the `JSON.toJSONString` method, ensuring its ability to seamlessly serialize Java timestamps into JSON strings. However, the code changes in the bug-inducing commit, as shown in Listing 2, extend the functionality of the `write` method to support nanosecond timestamps.

```
1     @Test
2     public void test_for_issue() throws Exception {
3         Timestamp ts = new Timestamp(Calendar.getInstance().
              getTimeInMillis());
4         String json = JSON.toJSONString(ts, SerializerFeature.
              UseISO8601DateFormat);
5         System.out.println(json);
6     }
```

Listing 1: Code snippet for testing the serialization of a Java Timestamp object to a JSON string

```
1  public void write(...
2  -    if (millis != 0) {
3  +    if (nanos > 0) {
4  +        buf = "0000-00-00_00:00:00.000000000".toCharArray();
5  +        int nanoSize = IOUtils.stringSize(nanos);
6  +        IOUtils.getChars(nanos, 30 - (9 - nanoSize), buf);
7  +        IOUtils.getChars(second, 19, buf);
8           ...
9  +    } else if (millis != 0) {
10          buf = "0000-00-00T00:00:00.000".toCharArray();
11          IOUtils.getChars(second, 19, buf);
```

Listing 2: Code changes to support nanosecond timestamps

However, an erroneous numeric boundary value "30" at Line 6 can trigger a `ArrayIndexOutOfBoundsException` whenever the subsecond part of the timestamp exceeds 100 nanoseconds. Due to the unstable nature of this test, reverting the code change elements other than those in Listing 2 has the opportunity to pass the test, causing DD to exclude the actual critical changes from its search space.

Even if we have tried to exclude bug entries that are related to obvious flaky tests, not all flaky tests can be easily identified and excluded. Flaky tests are a challenge to DD algorithms, which is out of the scope of this paper.

Finally, we identify three primary reasons from the *Worse* cases.

- Missing dependencies (77.47%): Due to the limited capabilities of DDJ, a considerable amount of dependencies are missing. Therefore, C2D2 has to struggle in detecting these intricate connections, which usually causes a time out.
- Differences between the code change elements generated by AST-based differencing techniques and those comprehended by humans (20.37%): C2D2 depends on the code change elements detected by Diff/TS, an AST-based differencing tool. However, Diff/TS tends to aggregate a set of edit operations as much as possible and may produce larger code change elements than those comprehended by humans. For example, a series of node deletions would be regarded as a single deletion of the subtree, which could be larger than the change elements observed by humans. In this case, the reduction results are still *technically* optimal and can be further improved by applying advanced code differencing techniques to generate the input of the DD process.
- Imprecise dependencies disclosed by the dependency matrix (2.16%): There are cases where the probability values in the dependency matrix do not reflect the actual dependencies. In such cases, imprecise dependency probabilities may lead to suboptimal results or timeouts due to lengthy attempts to recover complicated dependencies.

Note that most of the *Worse* results are caused by timeout, which means that no result is produced and thus no erroneous critical changes will affect the data quality when constructing a bug dataset. Only a very small part of the cases, mostly falling under the third reason, are actual data quality threats to the bug dataset.

We would also like to mention that, as the purpose of C2D2 is to construct a large-scale and high-quality bug dataset, the accuracy of the critical changes of bugs and the inclusion of bugs with complex changes is essential for the quality and diversity of the datasets. Full automation is equally critical for expanding the scale of dataset. Although the current version of C2D2 *can* also be used to extract critical changes during software development, the high cost of time and limited successful rate still remain an obstacle for daily debugging tasks.

Answer to RQ3: C2D2 is useful for extracting high-quality critical changes when used to construct large-scale bug datasets. It also outperforms the state-of-the-art tool for constructing bug datasets.

4.5 Threats To Validity

We briefly discuss the following three main threats to the validity of our evaluation.

- **Threats from Randomness:** Both ProbDD and C2D2 algorithms involve a degree of randomness, which can potentially impact various metrics in the experiments. To mitigate this threat, we ran the above methods and their variants three times and reported their averages as the final results of the experiments.

- **Threats from Data Quality:** The regression bugs used in the experiments were automatically generated by RegMiner and were not manually verified for accuracy. Incorrect data could pose a threat to the validity of the experiments. Therefore, we carefully filtered out duplicate or data with unstable tests to reduce its threat to the experiments.
- **Threats from Hyperparameters:** The settings of hyperparameters in both ProbDD and C2D2 could pose a threat to validity of the evaluations. To mitigate this threat, we conducted hyperparameter sensitivity experiments for both ProbDD and C2D2 and selected the optimal parameter values as the settings for evaluations. Details on the hyperparameter experiments can be found on our website.

5 RELATED WORK

Delta Debugging. The foundational Delta Debugging (DD) algorithm [46], denoted as *dd*, was introduced by Zeller to locate the minimal failure-inducing changes between two program versions. Later, Zeller proposed an extended version of *dd*, called *ddmin* [47], which isolates the minimal failure-inducing test inputs, and suggested its use over *dd*.

Since then, *ddmin* has been applied to various specific domains. Misherghi and Su [36] introduced HDD to enhance DD efficiency for tree structures. Subsequent works, such as modernized HDD [16], coarse HDD [17] and HDDr [21], were proposed to further improve HDD's performance. Heo et al. [15] presented CHISEL for code reduction in the C programming domain. CHISEL uses reinforcement learning to select steps in *ddmin* that are more likely to satisfy target properties, thereby enhancing *ddmin*'s effectiveness.

Different from *ddmin*, ProbDD [44] is a novel DD algorithm that learns from testing history the probabilities of the elements to be selected and employs a greedy algorithm to select elements based on the probabilities for each step, achieving significantly improved performance. Thus, ProbDD can be used to replace *ddmin* in both HDD and CHISEL. Similarly, C2D2 is also a novel dependency-sensitive DD algorithm. CHISEL is the closest to our work, which uses statistical models to enhance *ddmin* and ProbDD. CHISEL does not provide new selections of elements, but solely depends on *ddmin* and ProbDD for step generation. In contrast, C2D2 is capable of generating new choices in element selection and handling compilation errors.

Bug Dataset Construction. Bug datasets serve as foundational artifacts for various SE/PL tasks, such as software testing [3, 11, 12, 28, 29, 33, 35, 38], bug localization [4, 9, 22, 30, 31, 48], and bug repair [2, 7, 24, 37, 41]. Do et al. [6] are among the pioneers in constructing bug datasets, contributing the SIR dataset. Subsequently, a multitude of research efforts construct bug datasets from programming assignments and competitions (such as Marmoset [40], QuixBugs [27], IntroClass [8], and Codeflaws [42]), and open-source projects (such as BugBench [32], Corebench [1], BugJS [10], and DbgBench [2]).

These datasets are constructed manually, significantly impacting their scalability and representativeness. Dallmeier et al. [5] made the first attempt at semi-automated bug extraction by analyzing issues and associated commits. Afterwards, BEARS [34] and BugSwarm [43] automatically extract reproducible bugs from continuous integration systems. RegMiner [39] extracts bug-fixing commits and bug-inducing commits for regression bugs from code evolution history. However, these automated approaches do not exclude bug-irrelevant changes in commits. Defects4J [20] automates the extraction of bugs and test cases from bug reports and associated commits, but the critical changes of bugs were manually isolated.

BugBuilder [19] uses existing refactoring detection tools to detect and exclude refactoring changes from bug-fixing commits before enumerating all possible modification combinations to search for the minimized critical changes. It employs an exhaustive approach and does not work effectively on complex tasks.

DDJ [14] employs syntax- and semantics-aware techniques to group code changes and relies on *ddmin* to identify critical changes in Defects4J, with its performance affected by efficiency of *ddmin*.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we propose C2D2, a dependency-sensitive DD approach to extract critical changes for real-world bugs. We introduce a dependency matrix that tracks and estimates the probabilities of dependencies between code change elements. A matrix-based searching mechanism is proposed to attempt resolutions for compilation errors during the DD process. C2D2 also integrates sophisticated hybrid mechanisms for code change element selection during the DD process to search for optimal reduction results. Our evaluations on the Defects4J dataset and the regression bug dataset have confirmed the effectiveness and efficiency of C2D2.

In future, we plan to mitigate the risk of timeouts introduced by extra compilation attempts. We also plan to use C2D2 to enlarge existing bug datasets and try to categorize bugs based on critical bug-inducing and bug-fixing changes to allow new research opportunities for data-driven bug-fixing.

7 DATA AVAILABILITY

We have publicly released the source code and experimental results of C2D2, along with the benchmarks used in the experiments, on https://github.com/SongXueZhi/c2d2.

ACKNOWLEDGEMENT

The authors sincerely thank anonymous reviewers for their valuable comments and helpful suggestions. This work is partially supported by the National Science Foundation of China (62172099).

REFERENCES

[1] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: studying complexity of regression errors. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 105–115.

[2] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 117–128.

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 123–133.

[4] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005,*

St. Louis, Missouri, USA, Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (Eds.). ACM, 342–351.

[5] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer (Eds.). ACM, 433–436.

[6] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empir. Softw. Eng.* 10, 4 (2005), 405–435.

[7] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, Franz Rothlauf (Ed.). ACM, 947–954.

[8] Claire Le Goues, Neal J. Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Software Eng.* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[9] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 263–272.

[10] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 90–101.

[11] Mark Harman and Phil McMinn. 2010. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Trans. Software Eng.* 36, 2 (2010), 226–247.

[12] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2010. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical Software Engineering and Verification - International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures (Lecture Notes in Computer Science, Vol. 7007)*, Bertrand Meyer and Martin Nordio (Eds.). Springer, 1–59.

[13] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, Ahmed E. Hassan, Andy Zaidman, and Massimiliano Di Penta (Eds.). IEEE Computer Society, 279–288. https://doi.org/10.1109/WCRE.2008.44

[14] Masatomo Hashimoto, Akira Mori, and Tomonori Izumida. 2018. Automated patch extraction via syntax- and semantics-aware Delta debugging on source code changes. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 598–609.

[15] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 380–394.

[16] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, Tanja E. J. Vos, Sigrid Eldh, and Wishnu Prasetya (Eds.). ACM, 31–37.

[17] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 194–203.

[18] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. 2019. Deep transfer bug localization. *IEEE Transactions on software engineering* 47, 7 (2019), 1368–1380.

[19] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 686–698. https://doi.org/10.1109/ICSE43902.2021.00069

[20] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[21] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical Delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.). ACM, 16–22.

[22] Amy J. Ko and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 301–310. https://doi.org/10.1145/1368088.1368130

[23] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension*. 218–229.

[24] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 213–224.

[25] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 511–523.

[26] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[27] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Gail C. Murphy (Ed.). ACM, 55–56. https://doi.org/10.1145/3135932.3135941

[28] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1068–1080.

[29] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 440–451.

[30] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jin Song Dong. 2018. Break the dead end of dynamic slicing: localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 509–519.

[31] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jin Song Dong. 2017. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 393–403.

[32] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. BugBench: Benchmarks for Evaluating Bug Detection Tools. (01 2005).

[33] Kasper Søe Luckow, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 442–459.

[34] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 468–478.

[35] Phil McMinn. 2004. Search-based software test data generation: a survey. *Softw. Test. Verification Reliab.* 14, 2 (2004), 105–156. https://doi.org/10.1002/stvr.294

[36] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 142–151.

[37] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 772–781.

[38] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 778–788.

[39] Xuezhi Song, Yun Lin, Siang Hwee Ng, Yijian Wu, Xin Peng, Jin Song Dong, and Hong Mei. 2022. RegMiner: towards constructing a large regression dataset from

code evolution history. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 314–326. https://doi.org/10.1145/3533767.3534224

[40] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William W. Pugh. 2005. Software repository mining with Marmoset: an automated programming project snapshot and testing system. *ACM SIGSOFT Softw. Eng. Notes* 30, 4 (2005), 1–5.

[41] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 471–482.

[42] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 180–182. https://doi.org/10.1109/ICSE-C.2017.76

[43] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 339–349.

[44] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 881–892. https://doi.org/10.1145/3468264.3468625

[45] Andreas Zeller. 1999. DD.py: Enhanced Delta Debugging class. https://www.st.cs.uni-saarland.de/dd/DD.py. Retrieved August 4, 2023.

[46] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[47] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.

[48] Yao Zhang, Xiaofei Xie, Yi Li, Yun Lin, Sen Chen, Yang Liu, and Xiaohong Li. 2023. Demystifying Performance Regressions in String Solvers. *IEEE Trans. Software Eng.* 49, 3 (2023), 947–961.

Received 16-DEC-2023; accepted 2024-03-02