RegMiner: Towards Constructing a Large Regression Dataset from Code Evolution History

Xuezhi Song†
Fudan University
China
songxuezhi@fudan.edu.cn

Yun Lin∗
Shanghai Jiao Tong University
National University of Singapore
China/Singapore
dcslyn@nus.edu.sg

Siang Hwee Ng
National University of Singapore
Singapore
sianghwee@u.nus.edu

Yijian Wu†
Fudan University
China
wuyijian@fudan.edu.cn

Xin Peng‡
Fudan University
China
pengxin@fudan.edu.cn

Jin Song Dong
National University of Singapore
Singapore
dcsdjs@nus.edu.sg

Hong Mei
Peking University
China
meih@pku.edu.cn

ABSTRACT
Bug datasets lay significant empirical and experimental foundation for various SE/PL researches such as fault localization, software testing, and program repair. Current well-known datasets are constructed manually, which inevitably limits their scalability, representativeness, and the support for the emerging data-driven research.

In this work, we propose an approach to automate the process of harvesting replicable regression bugs from the code evolution history. We focus on regression bugs, as they (1) manifest how a bug is introduced and fixed (as non-regression bugs), (2) support regression bug analysis, and (3) incorporate more specification (i.e., both the original passing version and the fixing version) than non-regression bug dataset for bug analysis. Technically, we address an information retrieval problem on code evolution history. Given a code repository, we search for regressions where a test can pass a regression-fixing commit, fail a regression-inducing commit, and pass a previous working commit. We address the challenges of (1) identifying potential regression-fixing commits from the code evolution history, (2) migrating the test and its code dependencies over the history, and (3) minimizing the compilation overhead during the regression search. We build our tool, RegMiner, which harvested 1035 regressions over 147 projects in 8 weeks, creating the largest replicable regression dataset within the shortest period, to the best of our knowledge. Our extensive experiments show that (1) RegMiner can construct the regression dataset with very high precision and acceptable recall, and (2) the constructed regression dataset is of high authenticity and diversity. We foresee that a continuously growing regression dataset opens many data-driven research opportunities in the SE/PL communities.

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

KEYWORDS
mining code repository, bug collection, regression bug

ACM Reference Format:

1 INTRODUCTION
Bug datasets are fundamental infrastructure to support various software engineering researches such as software testing [7, 18, 19, 35, 36, 41, 43, 47], fault localization [9, 16, 29, 37, 38, 67], and bug repair [6, 13, 31, 46, 54]. The SE/PL community has taken decades to construct bug datasets such as SIR [12], BugBench [40], Codeflaws [55], and QuixBugs [34], finally gravitating to the state-of-the-art, Defects4J [26], which takes seven years to collect over 800 bugs.

While bug datasets like Defects4J [26] and CoREBench [5] have made significant contribution to the community, there are still concerns that their sizes are relatively small (and, thus arguably less representative [52]), regarding (1) their diversity (e.g., bug types of concurrency, API misuse, non-deterministic, etc.) and (2) the needs of applying the emerging data-driven and AI techniques
in SE tasks [15, 23, 39, 50, 51]. However, comparing to the cost of labelling an image or a sentence in the AI community, manually collecting and labelling a bug is far more expensive. It requires (1) prepare at least two versions of code project (i.e., a correct version and a buggy version), (2) set up at least one test case passing the correct version and failing the buggy version, and (3) isolate an environment where the bug can be well replicated. Manually constructed bug datasets [5, 12, 22, 26, 33, 34, 40, 55] naturally limit its scalability.

Recent works such as BEARS [42], BugSwarm [56], and BugBuilder [24] are emerged to automate the construction of bug datasets. BEARS [42] and BugSwarm [56] collect reproducible bugs by monitoring the buggy and patched program versions from Continuous Integration (CI) system. BugBuilder [24] is further designed to isolate bug-fixing changes in a bug-fixing revision. They can generally generate a bug dataset similar to Defects4J, with a much larger size. In such bug datasets, the buggy code is labelled with their fixes, potentially facilitating various data-driven bug-related learning tasks.

In this work, we further propose an approach, RegMiner, to harvest regression bugs from the code repository. Regression bugs are bugs which make an existing working function fail. Comparing to general bug analysis (e.g., bug repair and localization) which helps to derive a passing version from a buggy version, regression bug analysis helps to derive a passing version from a buggy version and a previously passing version. Typical regression analysis, such as delta-debugging [20, 21, 28, 45, 64], reports the failure-inducing changes from a large number of changes between the buggy and the previous passing version. The different problem setting makes Defects4J-like bug dataset hardly be useful for regression analysis. A large regression bug dataset has the following two benefits.

(1) Fewer Problems of Missing Specifications: Software bugs are essentially the inconsistencies between the implementation and its specification. Thus, a bug-related solution such as debugging and repair can be less reliable without sufficient specification as the input. In contrast to non-regression bug dataset where each bug is manifested with a few failing test cases, regression bugs are additionally equipped with their past passing version which can be used to cross-validate the violated specification with the (future) passing version, providing a more informative semantics to explain why the bug happens.

(2) Scalable Benchmark for Regression Analysis: From the perspective of benchmark construction, a large regression dataset can lay a foundation for various regression analysis. Existing regression analysis works (e.g., regression localization) are usually evaluated with limited number of regressions, given that their collection is highly laborious. Our investigation on 14 research works from the year of 1999 to 2021 [3, 8, 10, 27, 48, 54, 58, 60–64, 66] shows that the mean number of evaluated real-world regressions is 16.7, the median is 12.5, and the maximum is only 40. Moreover, different benchmarks are used in different work, making it difficult to compare their performance. A large-scale benchmark can not only mitigate the issue, but support various systematic empirical studies on regressions.

In this work, we design RegMiner to automate the regression harvesting process with zero human intervention. RegMiner can continuously harvest a large number of replicable regression bugs from code repositories (e.g., Github). Technically, we address an information retrieval problem in the context of software evolution history, i.e., retrieving runnable regressions on a code repository. Our approach takes a set of code repositories as input, and isolates a set of regressions with their running/replicable environment as output. Specifically, we construct a regression by searching in code repositories for a regression-fixing commit denoted as $rfc$, a regression-inducing commit denoted as $ric$, a working commit (before $ric$) denoted as $wc$, and a test case denoted as $t$ so that $t$ can pass $rfc$ and $wc$, and fail $ric$. To this end, we first design a measurement to select those bug-fixing commits with more regression potential. Next, for each such potential regression-fixing commit, $rfc$, we further search for a test case $t$ which can pass $rfc$, fail the commit before (denoted as $rfc$-1), and pass a commit further before $rfc$-1. Our search process addresses the technical challenges of (1) identifying relevant code changes in $rfc$ to migrate through the code evolution history, (2) adopting the library upgrades with the history, and (3) minimizing the compilation overhead and handling incomparable revisions.

We evaluate our approach with a close-world and an open-world experiment. In the close-world experiment, RegMiner achieves 100% precision and 56% recall on a benchmark consisting of 30 regression bugs and 50 non-regression bugs. In the open-world experiment, we run RegMiner on 147 code repositories within 8 weeks. RegMiner reports 1035 regressions which construct the largest Java regression dataset to the best of our knowledge. Our ablation study also shows the effectiveness and efficiency of our technical designs.

In conclusion, we summarize our contributions as follows:

- We propose a fully automated regression mining technique, which allows us to continuously harvest regressions from a set of code repositories with zero human intervention.
- We build our RegMiner tool with extensive experiments evaluating its precision and recall to mine regressions. The results show that RegMiner is accurate, effective, and efficient to mine regressions from code repositories.
- We construct a regression dataset with RegMiner within 8 weeks. We foresee that the size of our regression dataset can keep growing with time, opening the a number of research opportunities on bug analysis.

The source code of RegMiner is available at https://github.com/SongXueZhi/RegMiner. The mined regression dataset and its demonstration is available at https://regminer.github.io/.

2 PROBLEM DEFINITION AND REFORMULATION

Commit and Revision. A code commit can correspond to two revisions in the code repository, i.e., the revision before the commit and the revision after (or caused by) the commit. In this work, we use the terminology commit and its caused revision interchangeably. Thus, given a commit denoted by $c$, we also use $c$ to denote the revision caused by $c$. Moreover, we use $c - 1$ to denote the commit before $c$, i.e., the revision caused by the commit $c - 1$.

Regression. Given a fixed regression in the code repository, we denote it as $reg = (rfc, ric, f)$, which indicates that the regression $reg$ consists of a regression-fixing commit $rfc$, a regression-inducing commit $ric$, and a feature $f$, where (1) $f$ exists in both $rfc$ and $ric$.
and (2) $f$ works in $rfc$ and $ric$-1 but fails in $ric$. In addition, we call the commit $ric$-1 as the working commit (WC), and the feature $f$ as the regression feature.

**Problem Definition.** Formally, given a code repository $C$ as the a set of commits, we aim to construct a regression set $\mathcal{R} = \{\mathcal{R} \mid \mathcal{R} = (rfc, ric, f)\}$ where $\forall \mathcal{R} \in \mathcal{R}$, $\exists t.s.t. rfc \in C$, $ric \in C$, $rfc > ric$, $f$ works in $rfc$ and $ric$-1, and fails on $rfc$-1 and $ric$. The operator $>$ indicates the chronological partial order between two commits, and $rfc > ric$ indicates that the commit $rfc$ happens after the commit $ric$. The retrieval process of a fixed regression $reg$ is essentially the process of identifying $rfc$, $ric$, and $f$ in the code commit history.

While it is intuitive to look into a regression regarding its regression feature, the above definition still provides limited guidance for practical implementation. Specifically, we still need to answer the questions (1) how do we represent a regression feature? And more importantly, (2) how do we know that it is the same feature that works in one revision $reg_p$ in the past and fails in another revision $reg_f$ ($reg_f > reg_p$), especially when $reg_f$ has gone through many changes from $reg_p$?

**Problem Reformulation (for Practical Implementation).** In this work, we estimate a feature with a unit test with an oracle. The pass or failure of a unit test represents whether the feature works or fails. Therefore, we reformulate a regression as $reg = (rfc, ric, t)$ where $t$ is the feature-representing test. Practically, we search for (1) the test case $t$ and (2) a set of regressions $\mathcal{R} = \{\mathcal{R} \mid \mathcal{R} = (rfc, ric, t)\}$ where $\forall \mathcal{R} \in \mathcal{R}$, $\exists t.s.t. rfc \in C$, $ric \in C$, $rfc > ric$, $t$ passes on $rfc$ and $ric$-1, and fails on $rfc$-1 and $ric$. Moreover, we conservatively estimate that the feature represented by $t$ in $reg_p$ (i.e., passing revision) is compatible with that in $reg_f$ (i.e., failing revision) in practice if:

1. The methods tested by $t$ in $rfc$ and $ric$ are similar.
2. The assertion failures of $t$ in $rfc$-1 and $ric$ share the same root cause.

We adopt the conservative estimation because our regression mining approach generally favours precision over recall. Given the test $t$ in a regression $(rfc, ric, t)$, we also say that (1) $t$ is the regression test and (2) $t$ can compatibly pass $rfc$ and $ric$-1, and fail $rfc$-1 and $ric$.

## 3 OVERVIEW

We design our approach to mine and construct a regression dataset as Algorithm 1, which takes as input a set of code repositories and a similarity threshold to select potential regression-fixing commits. When searching for the regressions, our approach first collects the bug-fixing commits including test case addition from the code repositories (line 3-6). We further confirm a commit $c$ as bug-fixing if the added test case can pass $c$ and fail $c - 1$ (line 5-6), which serves as a prerequisite to search the regression bug. Then we estimate $c$’s potential as a regression-fixing commit (line 7). With the quantified regression potential, we can rank the bug-fixing commits and remove those with less potential (line 10). For each potential regression-fixing commit, we search for its regression-inducing commit $ric$ which satisfies that the test $test$ can compatibly fail in $ric$ and pass in $ric$-1 (line 12). Then, we record a regression $reg$ by $rfc$, $ric$, and the relevant test case $test$. The design of Algorithm 1 needs to overcome the following three challenges.

**Challenge 1: Futile Search on Non-regression Bug-fixing Commits.** Searching for a regression across the commit history is time-consuming, which includes the overhead of project compilation, test case migration, and test case execution. Starting with a non-regression-fixing commit causes the whole search futile. In this work, we design a novel measurement to quantify the potential of a bug-fixing commit to be a regression-fixing commit.

**Challenge 2: Test Dependency Migration.** With a test case and the potential regression-fixing commit, it is non-trivial to verify its regression-inducing commit. In practice, the regression can be induced years ago. The project can undergo radical changes and depend on different versions of libraries. Without appropriate adaptation, we can miss a large number of real regressions.

**Challenge 3: Large Overhead to Validate Regression.** Starting from a potential regression-fixing commit, there could be thousands of commits to check out, recompile, and run. Sequentially checking out revisions incurs huge runtime overhead, while bisect approach can suspend on some incommensurate commits.

### Algorithm 1: Regression Dataset Construction

<table>
<thead>
<tr>
<th>Input</th>
<th>: A code repository set, $repos$; a threshold of regression potential, $th_{reg}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>: A regression dataset, $reg$</td>
</tr>
</tbody>
</table>

```
// initialize the regression set
reg = {}  

for repo in repos do
    // initialize the regression set
    commits = search_commits_with_test_addition(repo)  
    for c in commits do
        // fixing commit confirmation
        is_fix = confirm_fix(c, test, c - 1)  
        if is_fix then
            // RFC prediction
            estimate_regression_potential(c)  
        else
            commits = commits \ {c}  
        end
    end
    commits' = rank_and_filter(commits, $th_{reg}$)
    for c in commits' do
        // search regression with test migration
        if c != null then
            rfc = c
            reg = (rfc, ric, test)
        end
    end
    regressions = regressions ∪ {reg}
end
```

return regressions

In practice, we consider two methods are similar if (1) their method names are the same and (2) the similarity between code bodies is above a threshold $th_{body}$ (e.g., 0.95).

We regard they share the same root cause if they share the same error message and error-occurring location in $t$.

**4 SEARCH METHODOLOGY**

In this section, we introduce how we address the three aforementioned challenges.
4.1 Estimating Regression Potential
Let us call the set of code elements implementing the regression feature as feature code. Assume that we know that (1) a feature f is fixed in a revision bfc, and (2) the precise set of code elements (e.g., methods) set_f of feature code, we can estimate the probability of bfc to be a regression-fixing commit as

$$P(rfc|bfc, set_f) = 1 - (1 - p)^N$$

(1)

In Equation 1, N is the number of changes applied on set_f in the commits before bfc. p is the probability that a change introduces a bug on the feature f, which we call as regression-inducing probability. The more changes applied on set_f in the history, the more likely the bfc is a regression-fixing revision.

However, it is non-trivial to precisely infer the program elements set_f purely based on a test case. Including irrelevant elements in set_f can make non-regression fixing revisions have a lower probability. Either case can make Equation 1 misguide the prediction of regression-fixing commit.

4.1.1 Feature Code Identification. We measure the relevance of a method m to a test case t by its uniqueness and similarity to t:

$$rel(m) = \min(1.0, test\_uniq(m, t) \times (1 + sim(m, t)))$$

(2)

In Equation 2, the relevance score rel(m) depends on (1) its test uniqueness (test_uniq(m, t)), i.e., how unique m is to t and (2) the textual similarity between m and t (sim(m, t)). We design test_uniq(m) and sim(m, t) with the range [0, 1]. The min(·) function in Equation 2 ensures that the relevance score has an upper bound of 1.0.

Test Uniqueness. To evaluate the test uniqueness of a method m for test, we design a variant TF-IDF measurement. Assume that there are N test cases in a project, and each test case has many methods in its call hierarchy. Thus, we can construct a test-method matrix Rmt where each column represents a test case and each row represents a method called by at least one test case. Specifically, we calculate an IDF-like measurement for each r(m, t) in Rmt as

$$test\_uniq(m, t) = r(m, t) = \frac{\log N}{\hat{freq}(m)} \quad \text{if } t \text{ covers } m$$

$$0 \quad \text{otherwise}$$

(3)

In Equation 3, N represents the number of test cases in the project, freq(m) represents the number of test cases covering method m, which ranges from 1 to N. Thus, logN/\hat{freq}(m) is scaled to the range [0, 1].

Textual Similarity. Moreover, we further introduce a similarity function between t and m. Specifically, we tokenize the name of test method into a bag of words with the word “test” removed, e.g., testCalendarTimeZoneRespected is converted into a bag as B = [“calendar”, “time”, “zone”, “respected”]. Let the number of tokens in the name/body of m which can match any of words in B be k, then the sim(test, m) = k/|B| which ranges between 0 and 1.

4.1.2 Code Element Re-identification. To collect historical modifications on a method, we need to re-identify them in the past commits. Given a method m in a revision r, if we can find a method m’ in another revision r’ with exact the same signature as m (r’ > r), we consider m’ as a match of m in r’. If we cannot locate such an exact match, we track their identity by defining their similarity:

$$sim(m, m’) = \alpha \cdot sim\_signature(m, m’) + \beta \cdot sim\_body(m, m’)$$

(4)

The metrics consist of the signature similarity and the code body similarity. We consider m and m’ as a match if there similarity is above a threshold thm. The similarity metrics are generally defined regarding method name similarity (the ratio of longest common subsequence over length of two method names), return types (same type or not), parameter types (Jaccard coefficient of two parameter type set), and method bodies (the ratio of longest common subsequence over length of two method bodies). Readers can refer to more details in our anonymous website [2].

4.1.3 Regression Potential Metric. Consequently, given a set of methods M, each m ∈ M has a relevance score denoted by rel(m) and a historical change number m.changes. Thus, we quantify the final probability of a regression-fixing commit as

$$P(rfc|bfc, set_f) = 1 - \prod_{m\in M} (1 - p \times rel(m))^{m.changes}$$

(5)

In Equation 5, p is the base empirical regression-inducing probability, e.g., 0.01, shared by all the methods.

4.2 Test Migration
Given a test t in a bug-fixing commit bfc, we need to migrate t along with its code dependency in a revision cinv under investigation (bfc > cinv). Typically, we need to overcome two challenges to avoid compilation errors:

- Identifying migrating dependencies. When we migrate t to the revision cinv, the code dependencies of t should be available in cinv.
- Reconciling migrated code. The migrated test and its dependencies can be adapted to the old libraries.

4.2.1 Identifying Migrating Dependencies. We consider a code element as either a class, interface, field, or method. We aim to find a minimum set of code elements in the bug-fixing revision bfc so that (1) the revision under investigation cinv migrated with the test t can still compile and run, and (2) the migration does not change the program behaviors of test case t under cinv. We achieve the former by parsing the minimum code elements depended by the regression test, and achieve the latter by identifying and refraining the migration of code elements with the bug-fixing changes. Figure 1 shows the workflow of identifying the code elements to be migrated, which consists of the following steps:

Step 1. Patch Analysis. We compare the revisions bfc (where t passes) and bfc-1 (where t fails) to locate the code elements in bfc where the fix happens. We denote such set of elements as Efix.

Step 2. Dependency Analysis. In bfc, we calculate the call graph rooted at the test case t (denoted as Ecg) and the set of covered elements by running t (denoted as Ecv). We keep the minimum dependencies in Ecg and Ecv.

Step 3. Code Alignment. Given bfc and the revision under investigation cinv, we detect how the code are aligned between two revisions. Thus, we can avoid migrating a code element matchable
in \( c_{inv} \). We use Xing and Stroulia’s UMLDiff algorithm \([59]\) to align the cross-revision code elements. As a result, we use \( E_{id} \) to denote the set of the identical elements between two revisions, and \( E_{mod} \) to denote the set of the modified elements (i.e., the elements matched in \( bfc \) with \( c_{inv} \) but with difference).

**Step 4. Migration.** Finally, the set of migrating elements is calculated as the missing dependencies \( E_{miss} = (E_{eq} \cup E_{ce}) \setminus (E_{fix} \cup E_{id}) \), and the changed dependencies \( E_{change} = (E_{eq} \cup E_{ce}) \setminus (E_{mod} \setminus E_{fix}) \). We will reconcile \( E_{mig} = E_{miss} \cup E_{change} \) on \( c_{inv} \). We will elaborate the reconciliation with more details in Section 4.2.2.

**Example.** Figure 2 shows simplified class dependency graphs of a regression-fixing commit \( bfc \) and a commit under investigation \( c_{inv} \) in Common Lang project. In Figure 2, we use yellow container boxes to represent classes (or interfaces), and containee boxes to represent class members such as method and field. The containment relations represent that a class/interface declares a method or a field. A method is a containee box whose name has "\( \)" as suffix, while fields are containee boxes without the suffix. The black arrows between the boxes are the call relation between methods and fields. We represent in green the elements covered by the test case in the bug-fixing commit.

In addition, the position of each box in \( bfc \) and \( c_{inv} \) represents the alignment relation between two revisions. For example, the class \( FastDataFormatter \) in \( bfc \) is aligned with the class with the same name in \( c_{inv} \). The missing depended elements by \( t \) in \( c_{inv} \) is represented by dashed boxes, the modified elements in \( c_{inv} \) is represented by red boxes, and the elements with code fix in \( bfc \) is represented by boxes with red border.

The workflow to identify the migration of code elements in \( bfc \) as follows.

1. **Patch Analysis.** In \( bfc \), we first identify the fixes by comparing \( bfc \) and \( bfc-1 \) to have \( E_{fix} = \{ FastDataFormatter.appendTo() \} \).
2. **Dependency Analysis.** Next, the static and dynamic dependency analysis from the test case \( testCalendarTimeZoneRespected() \), \( FastDataPrinter.format() \), \( DatePrinter.format() \), \( FormatCache.getStyle() \), and the changed dependencies as \( E_{change} = \{ FormatCache, cDateTimeInstanceCache \} \). Note that, the code elements with fix (i.e., \( FastDataFormatter.appendTo() \) ) is excluded.
3. **Code Alignment.** Then, we align two revision \( bfc \) and \( c_{inv} \) to know what dependencies of the test case are missing or changed. For example, we can know the missing dependencies such as \( FastDataPrinter.format() \) and \( DatePrinter.format() \). Also, we know that the field \( cDateTimeInstanceCache \) (in red) is modified and has a potential to reconcile in \( c_{inv} \).

![Figure 1: Identifying Dependency to be Migrated](image)

**Algorithm 2: Migration Reconciliation**

**Input:** missing dependencies, \( E_{miss} \); changed dependencies, \( E_{change} \); revision under investigation, \( c_{inv} \); regression test, \( test \);

**Output:** A migrated revision, \( c_{mig} \)

1. \( c_{mig} = migrate(E_{miss}, c_{inv}) \)
2. if compatible\( (c_{mig}, test) \) then
3. return \( c_{mig} \)
4. else \( c_{mig} = transform(c_{mig}) \)
5. if compatible\( (c_{mig}, test) \) then
6. return \( c_{mig} \)
7. else \( c_{mig} = transform(c_{mig}) \)
8. if compatible\( (c_{mig}, test) \) then
9. return \( c_{mig} \)
10. else
11. return \( c_{mig} \)
12. output null

**3. Migration.** Finally, we calculate the missing dependencies as \( E_{miss} = \{ testCalendarTimeZoneRespected(), FastDataPrinter.format(), DatePrinter.format(), FormatCache.getStyle() \} \), and the changed dependencies as \( E_{change} = \{ FormatCache, cDateTimeInstanceCache \} \). Note that, the code elements with fix (i.e., \( FastDataFormatter.appendTo() \) ) is excluded.

As for \( E_{mig} = E_{miss} \cup E_{change} \), we define transformation rules to adapt them in \( c_{inv} \).

**4.2.2 Reconciling Migrated Code.** We design our reconciliation heuristics with the following principles:

- **Missing Dependencies:** The code elements in \( E_{miss} \) are copied directly to \( c_{inv} \) without modifications. The copied \( E_{miss} \) will be adapted only if their migration incurs compilation errors on \( c_{inv} \).

- **Changed Dependencies:** The elements in \( E_{change} \) are not migrated unless their migration can minimize the compilation errors caused by the migration.

**Algorithm 2** shows our design to reconcile migration, which takes as input the missing dependencies \( E_{miss} \), the changed dependencies \( E_{change} \), the revision under investigation \( c_{inv} \), and the regression test \( test \) and derives a migrated revision to compatibly run the regression test. Generally, **Algorithm 2** is designed as a decision tree where each decision node guides the follow-up operations. Specifically, the decision is on whether the migrated revision is compilable. In **Algorithm 2**, the conditions are denoted in line 2, 6, 10, and 14. If the revision under transformation conforms to the condition, we return it as the revision; otherwise, we proceed to the next branch until either we can find a revision satisfying the condition or the migration is aborted (line 16).

In **Algorithm 2**, we first only migrate missing dependencies \( E_{miss} \) (line 1). If not successful, we proceed to transform the migrated revision with our pre-defined AST rewriting rules.
The rewriting rules transform the code regarding the different library versions in $c_{inv}$ and $bcf$ (line 5). In the transformation rules, we set a list of triggers such as syntax change and library version change. Those rules are generally defined regarding the API changes between the versions. Equation 6 shows one of our defined rules, in the form of the operational semantics. The precondition (the denominator) indicates that the JDK versions are different, and an existing-to-be-migrated method ($m \in M_{mis}$) defined in an interface ($m \in M_{inf}$) has an annotation of "@Override" (@Override $\in$ m.ANN). The postcondition (the numerator) indicates that we remove the "@Override" for that method. For example in Figure 2, when detecting that the supported JDK version in $c_{inv}$ is 1.5 while that in $bcf$ is 1.7, we will remove the "@Override" annotation on format() method in FastDatePrinter.\(^3\) Readers may refer to [2] for more details.

$$
\begin{align*}
JDK_\text{v}(c_{inv}) &= 1.5, \quad JDK_\text{v}(bcf) > 1.5, \\
\exists m &\in M_{inf} \land m \in M_{mis}. \quad "@Override" \in m.ANN \\
&\quad \quad \quad m.ANN \rightarrow m.ANN \setminus \{"@Override\}\quad (6)
\end{align*}
$$

We proceed to migrate the changed dependencies if the transformation is not successful (line 8). Different from migrating missing dependencies, we migrate the changed dependencies as an optimization problem. We aim to migrate and transform the minimum changed dependencies to derive a compatible revision (line 9 and 13). Specifically, we repetitively overwrite the code elements in $c_{inv}$ with elements in $E_{change}$ to search for a minimum core to satisfy the conditions. In the example of Figure 2, we will overwrite the field cDateTimeInstanceCache in $c_{inv}$ with that in $bcf$ for eradicating the compilation errors. Note that, the static method getPatternForStyle() invoking that field is migrated to $c_{inv}$, which leads to the change of the field’s modifier to static.

\(^3\)In JDK 1.5, the "Override" annotation is not allowed to describe a method implementing the interface

---

**Figure 2:** Migrating test dependencies to a revision.

**Figure 3:** Search feedback revisions forward and backward

### 4.3 Validation Effort Minimization

Given a bug-fixing commit, we validate it as a regression-fixing commit by searching for a regression-inducing commit $c$ in the history where the test case fails $c$ but passes $c-1$. A naive search algorithm can be a binary search algorithm as git bisect implementation [1]. The binary search algorithm assumes that each revision provides us with a feedback (such as test case pass or failure), which guides us to either approach the revision $c$ where $c$ can fail and $c-1$ can pass (i.e., regression bug), or the initial revision where $c$ just fails (i.e., non-regression bug).

However, the revisions cannot always be compilable in practice. Moreover, the test code and its dependencies migrated by RegMiner can also suffer from incompilability. In such a case, the potential compilation errors provide less guidance and the test case cannot be resolved to provide feedback. Therefore, when visiting a revision with no feedback during the search, we design an approach to either (1) search for the closest revision with guiding feedback to get rid of the "no-feedback region" in the history, or (2) quickly abandon the bug-fixing commit to proceed with the next one.

**Skipping No-feedback Region.** Figure 3 shows an example where we visit a no-feedback revision when searching over the code history. Our approach estimates the no-feedback region with an exponential search algorithm as Algorithm 3. Taking a no-feedback revision $v$ and a boundary revision as input $b$ (indicating the searched boundary region cannot be beyond $(v, b)$ or $(b, v)$), Algorithm 3 returns a revision $h_v$ between $v$ and $b$ so that $h_v$ can be either (1) a feedback revision closest to the no-feedback revision $v$ or (2) the
A boundary revision with feedback, \(b_r\),

\[\text{Output:} A \text{ boundary revision with feedback, } b_r\]

// the search direction is initialized as from \(v\) towards \(b\)
1 direction = \(v > b\)
2 cursor = \(\text{prev} = v\)
// the revision to be returned
3 \(b_r = b\)
// the search range is within \(b_{past}\) and \(b_{future}\)
4 \(b_{future} = \max(a, b), b_{past} = \min(a, b)\)
5 \(\text{step} = 1\)
6 while true do
   \(\text{prev} = \text{cursor}\)
   // move towards the direction with a step size
7 \(\text{cursor} = \text{move(cursor, step, direction, test)}\)
   // when we find a revision \(b_r\) which can first feedback
8 if \(\text{cursor has feedback and prev has not feedback then}\)
   \(b_r = \text{update_best(b_r, cursor)}\)
   direction = \(\neg\text{direction}\)
   \(\text{step} = 1\)
else if \(\text{cursor has feedback and prev has feedback then}\)
   \(\text{return cursor}\)
   \(b_r = \text{update_best(b_r, cursor)}\)
else if \(\text{cursor has not feedback and prev has feedback then}\)
   \(\text{if cursor is out of boundary } (b_{past}, b_{future}) \text{ then}\)
   \(\text{return cursor}\)
else if \(\text{cursor has not feedback and prev has feedback then}\)
   \(\text{if cursor is out of boundary } (b_{past}, b_{future}) \text{ then}\)
   \(\text{return cursor}\)
else if \(\text{cursor has not feedback and prev has feedback then}\)
   \(\text{cursor} = \text{prev}\)
   \(\text{step} = 1\)
23 \(\text{step} = \text{step} \times 2\)

Algorithm 3: Closest Feedback Revision Search

Input: A no-feedback revision, \(v\); a code repository, repo; a test case, test; boundary revision \(b\)

Output: A boundary revision with feedback, \(b_r\)

Algorithm 4: Regression Introduction Search

Input: A revision, head; A revision, tail; a code repository, repo; a test case, test

Output: A regression-inducing revision, \(r_{ic}\)

\[v = \text{repo.get_middle}\]
1 \(v = \text{repo.get_middle}\)
2 \(\text{while} \ tail > head \ do\)
3 \(\text{if} \ v \ has \ feedback \ then\)
4 \(\text{head} = v\)
5 else
6 \(\text{tail} = v\)
7 else
8 \(b_1 = \text{search_feedback_revision}(v, \text{repo, test, head})\)
9 \(b_2 = \text{search_feedback_revision}(v, \text{repo, test, tail})\)
10 if \(\text{head} == b_1\) \&\& \(\text{tail} == b_2\) \then
11 \(\text{return null}\)
12 if \(b_2\) \can fail \then
13 \(\text{head} = b_2\)
14 \(\text{if} \ b_1\) \can fail and \(b_2\) \can fail \then
15 \(\text{head} = b_1\)
16 \(\text{tail} = b_2\)
17 \(\text{if} \ b_1\) \can fail and \(b_2\) \can fail \then
18 \(\text{tail} = b_1\)
19 \(v = \text{repo.get_middle}\)
20 \(v = \text{repo.get_middle}\)
21 \(\text{return head} + 1\)

5 EXPERIMENT

We build RegMiner to mine Java regressions, supporting maven and gradle projects. We evaluate RegMiner with the following research questions, more details of our experiment are available at our anonymous website [2].

- **RQ1 (Close-world Experiment):** Can RegMiner accurately and completely mine regressions from Git repositories?
- **RQ2 (Open-world Experiment):** Can RegMiner continuously mine authentic regressions from real-world Git repositories? How diverse is the regression dataset?
- **RQ3 (Ablation Study):** How does each technical component in RegMiner contribute to the mining effectiveness and efficiency?

5.1 Close-world Experiment

5.1.1 Experiment Setup. We manually collect 50 regression-fixing commits and 50 non-regression-fixing commits from Github repositories for the measurement of precision and recall. To avoid bias, we construct the regression benchmark without any help of RegMiner. Specifically, we search for closed Github issues each of which uniquely mentions a commit as its solution. Then, we filter those issues based on its labels and description. We prioritize the issues with labels as "regression" or "bug", or with description with the keyword of "regression". Then, we confirm the real regressions by manually checking the evolution history for regression-inducing commits, and migrating the test cases. As a result, we confirmed 50 regressions from 23 Java projects (see [2] for more details). By similar means, we identified 50 non-regression bugs.
We apply RegMiner on all the regression and non-regression fixing commits to measure the precision and recall. Let the number of true regressions be $N$ ($N = 50$), the number of reported regressions be $M$, and the number of reported true regressions be $K$, the precision is $\frac{K}{N}$, the recall is $\frac{K}{M}$. We choose the empirical regression-inducing probability $p$ as 0.05.

### Results

RegMiner achieves 100% precision and 56% recall in the experiment. Since the number of selected non-regression fixing commit is small in this experiment, we leave the discussion on the potential false positives to Section 5.2.2. Next, we discuss the reasons of false negatives. Our investigation shows that the recall mainly suffers from three folds: (1) our engineering implementation has not supported some sophisticated project configurations (16 out of 22), (2) insufficient AST adaption (see Section 4.2) for test case migration (5 out of 22), and (3) our validation effort minimization can sometimes miss the regression-inducing commit (1 out of 22).

**Limited Engineering Support and Test Dependency Migration.** Our investigation on the false negatives reveals that the realization of our methodologies requires more sophisticated implementation beyond our current engineering support. RegMiner still has its limitation under the scenarios such as the project-build configuration beyond maven and gradle, specific maven plugin (e.g., maven license plugin can stop the project building if our migration does not include licence information in the source code), and the dependency on GCC to compile the project. Moreover, RegMiner also has its limitation to handle complicated Java syntax like lambda expressions or `enum` classes. We will strengthen our engineering support to generalize RegMiner in more practical scenarios.

**Validation Effort Minimization.** We observe that, the efficiency pursued by our VEM (Validation Effort Minimization) technique, sometimes can scarify the accuracy, leading to the false negatives. Figure 4 shows an example. In Figure 4, we show the visiting order of RegMiner on one ground truth regression (univoc city-parsers project with fixing commit e48cdc). In this case, RegMiner visits non-feedback revisions in all the first four steps. Based on Algorithm 4, RegMiner tries to search for a feedback revision by exponentially increasing the search step size (line 23 in Algorithm 3). A mitigation is to change the exponential increase to a less radical increase approach, e.g., increase the moving step by a fix size. Nevertheless, it will incur large overhead as a tradeoff. We leave the choice to the practitioners in their own applications.

### 5.2 Open-world Experiment

#### 5.2.1 Experiment Setup

In this experiment, we collect Git repositories from Github and run RegMiner to mine regressions for 8 weeks. We evaluate RegMiner regarding the authenticity and the diversity of the mined regressions in both quantitative and qualitative manner.

**Quantitative Analysis.** We quantitatively estimate how likely the tested feature in the regression-fixing revision (RFC) exists in the regression-inducing revision (RIC), based on the rationale that the feature in RFC is likely to be preserved in RIC if the test covers similar set of code elements in both revisions. Specifically, for each mined regression $\text{reg} = \langle \text{rfc}, \text{ric}, t \rangle$, we run $t$ against $\text{rfc}$ and $\text{ric}$ to have their covered method set $E_{\text{rfc}}$ and $E_{\text{ric}}$. We consider $m_f \in E_{\text{rfc}}$ can be aligned with $m_i \in E_{\text{ric}}$ if UMLDiff algorithm [59] can match them. Therefore, we measure the feature coverage similarity in rfc and ric by:

$$\text{sim}_f(\text{reg}) = \frac{|E_{\text{rfc}} \cap E_{\text{ric}}|}{\min(|E_{\text{rfc}}|, |E_{\text{ric}}|)}$$

**Qualitative Analysis.** We further sample 30 regressions with high coverage score (>0.9), 30 with medium score (0.4-0.7), and 30 with low score (<0.1). To facilitate the manual investigation, we build a tool to visualize and compare revision difference, run test cases against each revision, and revert changes between revisions. A screenshot of the tool is available at our website [2]. We recruit two graduate students (both with over 5 years of Java programming experience) to manually verify each regression based on the following template questions:

- What is the feature in RFC aims to fix?
- What is the reason of the regression bug?
- Why can the feature work well before RIC?

The two students were asked to write down their answer independently. Then, they are further asked to discuss with each other to reconcile their answers. Given a regression, if they agree that the feature tested in RFC is different from the feature tested in RIC, we conclude the regression is a false positive.

**Diversity Analysis.** We measure the diversity of the regression dataset by investigating the diversity of (1) topics of used libraries of the regression and (2) the exception types of each bug. Assume the regressions are distributed in K library topics, and reported with L exception types (e.g., `NullPointerException`), we compare the diversity between our collected datasets and Defects4j on (1) the number of library topics and reported exception types, and (2) the information entropy of bug distribution on the topics and exception types.

#### 5.2.2 Results

In this experiment, we explore 30,165 commits in which 6,253 commits are bug-fixing commits. As a result, RegMiner constructs a regression dataset consisting of 1035 regressions over the 147 projects within 8 weeks. Among the 1035 mined regressions, over 72% commits incur runtime exceptions when the test is migrated to a past commit. Nevertheless, RegMiner still manages to report them with the validation effort minimization technique (see Section 4.3).
unnecessarily escape special character '?', '&'
the escape function has not been implemented yet

Figure 6: False positive reported by RegMiner

Listing 1: A regression test for checking whether special character in HTML can be handled well

```java
@Test
public void strictAttributeKsesapes() {
  String html = "<a href="/foo/bar/&lt;true">Ok</a>";
  Elements els = Jsoup.parse(html).select("a");
  assertEquals("?foo=bar&mid&lt=true", els.first().attr("href"));
}
```

Figure 5: Feature Coverage Similarity Distribution

Diversity Analysis. Table 1 shows that the regressions collected by RegMiner are more diverse than Defects4j regarding the number of libraries, the involved library topics, and the types of exceptions. Comparing to Defects4j, regressions in RegMiner use 7.3X libraries, cover 3.5X topics, and raise 1.2X more types of exceptions. More details of Venn Diagram to compare the diversity of RegMiner and Defects4j is available at [2].

<table>
<thead>
<tr>
<th>#Prj</th>
<th>Library Diversity</th>
<th>Topic Diversity</th>
<th>Exception Diversity</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegMiner</td>
<td>147</td>
<td>818</td>
<td>110</td>
</tr>
<tr>
<td>Defects4j</td>
<td>17</td>
<td>122</td>
<td>31</td>
</tr>
</tbody>
</table>

Overall, our conservative strategy (i.e., the strict conditions to report a regression regarding test compatibility on the passing and failing revisions) lead to high precision, at the cost of moderate recall. In our future work, we will study RegMiner’s improvement on recall (see Section 6 for more details).

5.3 Ablation Study

5.3.1 Experiment Setup. We disable, replace, and enable the three components respectively to evaluate their effectiveness.

Regression Potential. Given that the number of regression fixing commits is $N$ ($N = 1035$), we randomly select another $N$ bug fixing commits. Then, we let RegMiner generate regression potential metric for each fixing commit, comparing the metrics distribution of the two groups. Moreover, we further evaluate our regression potential metrics by investigating the effectiveness of its rank on the regression-fixing commits. Specifically, given $N$ regression-fixing commits and $aN$ ($a > 1$) non-regression fixing commits, we evaluate how many regression-fixing commits RegMiner can report by looking the first $k\%$ reported commits. We let $a = 2$ because that non-regression fixing commits are usually more than regression-fixing commits. We choose $k = 2$ in this study.

Test Dependency Migration and Validation Effort Minimization. In this study, we disable and simplify the capability of test dependency migration and validation effort minimization separately.

- **Disabled Test Dependency Migration (RegMiner −$TD_M$):** We have RegMiner −$TD_M$ for RegMiner with test dependency migration disabled. In RegMiner −$TD_M$, we copy the identified test case into the target revision without dependency analysis.
- **Simplified Validation Effort Minimization (RegMiner −$V_E M + b i s e c$):** We have RegMiner −$V_E M + b i s e c$ for RegMiner with validation effort minimization simplified. In RegMiner −$V_E M + b i s e c$,
we search for regressions by using git-bisect strategy [1]. Moreover, if RegMiner $\neg_{VEM+bisec}$ visits a revision $v_{fail}$ without any feedback, we conservatively estimate a failure feedback on $v_{fail}$.

- **Disabled TDM and Simplified VEM (RegMiner $\neg_{VEM+TP}$):** We have RegMiner $\neg_{TDM+TP}$ for RegMiner with test dependency migration disabled and search for regression with git-bisect strategy.

- **Disabled TDM and Simplified VEM (RegMiner $\neg_{TP+git blame}$):** We have RegMiner $\neg_{TDM+git blame}$ for RegMiner with test dependency migration disabled and search for regression with git-blame strategy on the difference between $rfc$ and $rfc-1$. We consider the strategy on a regression successful if (1) git-blame can report $ric$ and (2) the test migration from $rfc$ to $ric$ is successful.

We compare RegMiner with the aforementioned variants on the datasets in both close-world and open-world experiments. Given the open-world experiment takes 8 weeks in this study, we replicate the experiment with the following simplification. Given RegMiner reports $N$ regressions in the open-world experiment ($N=1035$ in this study), we randomly sample $2 \times N$ bug-fixing commits, and run RegMiner, RegMiner $\neg_{TDM}$, and RegMiner $\neg_{VEM}$ on the $3 \times N$ bug fixing commits to observe their performance.

5.3.2 Results.

Regression Potential Estimation. We evaluate the regression potential scores on the 1035 regression-fixing commits and 1035 randomly chosen bug-fixing commits. Overall, the regression-fixing group has an average of 0.51 and median of 0.43; in contrast, the bug-fixing group has an average of 0.38 and median of 0.30. We apply unpaired two-samples Wilcoxon test on two groups and have the $p$-value smaller than $10^{-5}$, indicating that the regression-fixing group is significantly different from the bug-fixing group. In addition, Figure 7 shows the effectiveness of RegMiner to select the regression-fixing commits. Overall, the first 20% ranked commits can include over 50% regression-fixing commits (the blue curve in Figure 7), outperforming random selection (the orange curve in Figure 7). Thus, we conclude that our regression potential metric is effective.

Test Dependency Migration and Validation Effort Minimization. Table 2 shows the overall regression retrieval performance. Overall, comparing to the baselines, our solution shows its effectiveness. While achieving the leading performance, we also find that the variants such as RegMiner $\neg_{VEM+TP}$, sometimes can report the regressions missed by RegMiner. It is because different search strategy makes different tradeoffs, leading to different search results. In addition, simplified variants such as RegMiner $\neg_{TDM+TP}$ and RegMiner $\neg_{TDM+git blame}$ incur less runtime overhead, but have large degradation on recall.

Table 2: The overall performance of regression retrieval

<table>
<thead>
<tr>
<th>Approach</th>
<th>Close-world Experiment</th>
<th>Open-world Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>RegMiner</td>
<td>1.0</td>
<td>0.56</td>
</tr>
<tr>
<td>RegMiner $\neg_{TDM}$</td>
<td>1.0</td>
<td>0.52</td>
</tr>
<tr>
<td>RegMiner $\neg_{VEM+TP}$</td>
<td>1.0</td>
<td>0.47</td>
</tr>
<tr>
<td>RegMiner $\neg_{TDM+TP}$</td>
<td>1.0</td>
<td>0.04</td>
</tr>
<tr>
<td>RegMiner $\neg_{TDM+git blame}$</td>
<td>1.0</td>
<td>0.20</td>
</tr>
</tbody>
</table>

5.4 Threats to Validity

One internal threat is the precision (100%) reported in our close-world experiment, which is only reported on 50 regressions and 50 non-regression bugs. Given the small number of regressions, it may not be sufficient to conclude that the precision can be applied to the open-world experiment. To mitigate the issue, we further manually sample 90 regressions in the open-world experiment, to complement the precision evaluation. One external threat lies in that our implementation is based on Java projects, further studies on other popular programming language such as Python and C++ are still needed to generalize our findings.

6 DISCUSSION

Our experiments have shown that RegMiner can successfully construct a runnable regression dataset, with its size continuously growing, showing good potential as a research infrastructure to support various software engineering studies. In this section, we discuss the follow-up work regarding (1) improving the mining effectiveness, (2) realizing the potential of this infrastructure, and (3) its alternative applications.

6.1 Mining Effective (Recall) Improvement

We can improve the mining effectiveness by (1) on-the-fly rewriting rule generation, (2) regression test case selection, and (3) multiple-test migration.

On-the-fly Rule Generation. RegMiner uses manually defined AST-rewriting rules to reduce the compilation errors induced by test-case migration. In the future work, we will investigate how to derive semantic-preserving rules on-the-fly by comparing the different versions of library used in two commits.

Regression Test Selection. RegMiner only uses the test case created in the regression-fixing commit, assuming that regressions are fixed along with a complementary regression test. The conservative strategy may miss some regressions with their regression tests.
which have been created before the fixing-commit. We will expand the scope of regression-test detection to improve the recall. **Multiple-test Migration.** Migrating $N (N > 1)$ test cases potentially allows us to find multiple regressions by a single search. The challenge lies in how to deal with different feedbacks of different tests when searching for the regression-inducing commit. We will design a more optimal solution to maximize the "reward" of the search.

6.2 Dataset Quality
Since we intend to build a large dataset to evaluate various software testing, debugging, and repair research work, the dataset quality is important. Despite the chance of false positives is small, the open-world experiment still reveals such a possibility. Moreover, despite existing techniques such as delta-debugging [57, 64] can help RegMiner to isolate failure-inducing changes, the human effort to verify and annotate the changes is almost inevitable with the existing software engineering techniques.

Therefore, we call for and foresee a crowdsourcing platform for researchers and volunteers to contribute, with the shared RegMiner facility. The following questions need to be answered:

(1) **Social Aspect:** how to design a crowdsourcing system to involve the researchers in the community?

(2) **Tool Support:** how to design an intuitive bug/regression annotation tool to confirm the bug with minimum effort?

(3) **Technical Design:** how to improve existing software engineering techniques to recommend failure-inducing more precisely?

We designed a regression annotation prototype as the first step towards this direction (see [2]). The demonstration are available at https://regminer.github.io/.

6.3 Potential Beneficiary
RegMiner is intended for the researchers in the PL/SE community, which allows them to flexibly search for different types of bugs/regressions for their study. Nevertheless, we foresee that RegMiner can also be used by practitioners in software industry to build project-specific benchmark. Industrial practitioners can use RegMiner to (1) understand what project functionalities are more likely to introduce regression bugs, (2) who are more likely to introduce regression, and (3) collect reusable regression patches for future fix recommendation and reference. The first and second application can help project managers to make organizational decision, and the third application is useful for software developers to avoid reinventing the wheels.

7 RELATED WORK
7.1 Bug Dataset Construction
Researchers have proposed many bug datasets in the community. Hutchins et. al [22] construct the first real-world bug dataset. Do, Elbaum, and Rothermel [12] further construct the SIR dataset. Following their work, various datasets are constructed from programming assignments and competitions (e.g., Marmoset [53], QuixBugs [34], IntroClass [32], Codeflaws [55], etc.), open source projects (e.g., DbgBench [6], Defects4j [26], BugSJS [17], BugS.jar [49], etc.), and runtime continuous integration scenarios (e.g., BEARS [42] and BugSwarm [56]). The most relevant dataset CoREBench [5], which is a regression dataset of 70 C/C++ regressions.

Those bug datasets are constructed manually, which naturally affects the scalability and representativeness of the bug dataset. Dallmeier and Zimmermann [11] make the first attempt to construct bug dataset in a semi-automatic way, via analyzing bug issues and their commits. Zhao et. al [68] further propose to replicate bugs based on Android bug reports. Recently, BEARS [42] and BugSwarm [56] are proposed to construct the bug dataset by collecting the buggy and the patched version on Continuous Integration system. Following their work, Jiang et. al [25] further propose BugBuilder to construct dataset by isolating the bug-relevant changes. RegMiner is different in that we target for a regression bug dataset, and address new technical challenges such as regression-fixing commit prediction, test dependency migration, and validation effort minimization.

7.2 Regression Research
Regression research includes regression fault localization [8, 20, 21, 28, 45, 57, 58, 60, 64, 65], regression testing [4, 14, 30, 44], and regression explanation [58, 60]. Zeller [64] pioneers the delta debugging algorithm, which is followed by a number of variants in specific scenarios [20, 21, 28, 45]. The most recent delta-debugging variant is proposed by Wang et. al [57], as probabilistic delta debugging, to further lower the algorithm complexity while improve the accuracy. Tan and Roychoudhury [54] propose a regression bug repair technique by searching over the regression revision through predefined code transformation rules. As for regression explanation, Wang et. al [58] propose a state-of-the-art regression explanation technique by proposing a novel alignment slicing algorithm on the execution traces of the regression version and past working version. However, researchers usually either prepare their own dataset with limited size or inject mutated regressions which are less representative for the real-world regressions.

Our RegMiner solution can largely mitigate the challenges, laying foundation for the future regression analysis and motivating both empirical and experimental facilities for the follow-up regression research.

8 CONCLUSION
In this work, we propose RegMiner which can automatically construct regression dataset. We address the challenges of predicting regression-fixing commits, migrating test and its dependencies, and minimizing the regression validation effort. Our close-world experiment shows that RegMiner achieves acceptable recall. Our open-world experiment shows that RegMiner has constructed an authentic and diverse regression dataset within a short time.

ACKNOWLEDGEMENT
We sincerely thank anonymous reviewers for their comments to improve this work. This research is supported in part by the Ministry of Education, Singapore (T2EP20120-0019, T1-251R5S1901, MOET32020-0004), A*STAR, CISCO Systems (USA) Pte. Ltd and National University of Singapore under its Cisco-NUS Accelerated Digital Economy Corporate Laboratory (Award I21001E0002), and National Natural Science Foundation of China (62172099).