On the Reproducibility of Software Defect Datasets

Hao-Nan Zhu
University of California, Davis
United States of America
hnzhu@ucdavis.edu

Cindy Rubio-González
University of California, Davis
United States of America
crubio@ucdavis.edu

***Abstract*—Software defect datasets are crucial to facilitating the evaluation and comparison of techniques in fields such as fault localization, test generation, and automated program repair. However, the reproducibility of software defect artifacts is not immune to breakage. In this paper, we conduct a study on the reproducibility of software defect artifacts. First, we study five state-of-the-art Java defect datasets. Despite the multiple strategies applied by dataset maintainers to ensure reproducibility, all datasets are prone to breakages. Second, we conduct a case study in which we systematically test the reproducibility of 1,795 software artifacts during a 13-month period. We find that 62.6% of the artifacts break at least once, and 15.3% artifacts break multiple times. We manually investigate the root causes of breakages and handcraft 10 patches, which are automatically applied to 1,055 distinct artifacts in 2,948 fixes. Based on the nature of the root causes, we propose automated dependency caching and artifact isolation to prevent further breakage. In particular, we show that isolating artifacts to eliminate external dependencies increases reproducibility to 95% or higher, which is on par with the level of reproducibility exhibited by the most reliable manually curated dataset.**

***Index Terms*—software reproducibility, software defects, software maintenance, software quality**

I. Introduction

Reproducibility in software engineering means that a result from an experiment can be repeatedly observed by others following the same procedure and using the same software artifact [8]. Reproducibility is imperative for research [10]. The ability for software artifacts to build and execute as they did originally with the same end result is crucial. The inability to repeat or extend the results of an experiment with a software artifact would defeat its purpose. In particular, software defect datasets [13, 21, 29, 30, 33, 37, 48, 58] have a large impact on the software engineering research community. These datasets enable the study and design of new techniques for fields such as automated program repair [20, 61, 64, 65], test prioritization [34, 35, 39, 41, 44, 51], test generation [17–19, 28, 50, 63], and fault localization [22, 46, 52, 66]. These fields have helped make software more robust, efficient, and secure.

Software defect datasets contain hundreds of defects collected from real-world projects. These datasets are comprised with a buggy version of the source code and the corresponding fixed version. Additionally, the process of building and testing is automated, and does not require domain-specific knowledge. An important aspect of any such dataset is to provide a reliable way to successfully build its artifacts so that researchers can replicate experiments or conduct new studies. Without this, their adoption and usefulness would be limited.

A troubling fact is that software defect artifacts are not immune to breakages after their creation [55]. Breakage in this context means that the original defects and fixes are no longer reproducible. Software breakage in these artifacts is no different, and perhaps more prone. Such artifacts are historical versions of the software, which are more likely to suffer from missing dependencies or external resources that affect their build outcome. Additionally, software defect artifacts can become broken due to previously passing tests that start failing due to deprecated APIs, or unavailable resources required during the testing process. The inability to evaluate or study these artifacts hinders research progress by prohibiting comparisons in future evaluations. Surprisingly, to the best of our knowledge, there is no literature on the reproducibility of software defect artifacts and their proneness to breakages.

In this paper, we present a study on the reproducibility of software defect datasets. First, we conduct a study on reproducibility of five state-of-the-art Java software defect datasets: Defects4J [30], GrowingBugs [29], Bugs.jar [48], BugSwarm [58], and Bears [37]. We aim to answer two research questions: (1) What criteria and strategies do datasets apply to determine reproducibility and to reduce the risk of software breakage, and (2) How are datasets affected by breakages. We find that reproducibility has different meanings across datasets. For example, while some datasets consider the *existence* of any test failure sufficient to consider a bug reproducible, others require matching the *number* and *names* of failing tests. As for reproducibility strategies, different approaches are applied to ensure reproducibility, from reproducing multiple times upon artifact creation to managing project dependencies and using containerization. Furthermore, we find that software breakages affect all datasets, especially those that adopt automated approaches for their creation whose reproducibility ranges from only 26.6% to 69%.

Second, we conduct a case study on the reproducibility of the BugSwarm dataset. BugSwarm is designed to continuously grow, and adopts containerization to ensure reproducibility, i.e., each artifact is packed as a Docker image. This facilitates the study and the fix of breakages. The goal is to determine (1) How software defect artifacts experience breakages through time, (2) What are the root causes and corresponding patches for breakages, and (3) How to prevent future breakages to ensure long-term reproducibility. We systematically monitor

TABLE I
DATASETS FOR REPRODUCIBILITY STUDY. *Release* INDICATES THE EXACT VERSION PICKED FOR THE STUDY.

Dataset	# Projects	# Artifacts	Artifact Source	Build System	Bug Location	Provided Reference	Release
DEFECTS4J	17	864	Issue Tracker	Ant / Maven	Source	List of Failing Tests	`397075d`
GROWINGBUGS	150	570	Issue Tracker	Ant	Source	List of Failing Tests	`6071840`
BUGS.JAR	8	1,158	Issue Tracker	Maven	Source	Build Log for Buggy	`8410717`
BUGSWARM	120	1,795	Travis-CI	Ant / Gradle / Maven	Source / Config	CI Log for Buggy & Fixed	`181f304`
BEARS	72	251	Travis-CI	Maven	Source	List of Failing Tests	`912bb98`

the reproducibility of 1,795 Java artifacts over a 13-month period. We find that 1,124 (62.6%) artifacts break at least once, and 275 (15.3%) break multiple times. With a test interval of 11.7 days, on average 32 artifacts are newly broken during each test. To identify root causes of breakages, we manually inspect 1,606 breakages. We identify 11 root causes and handcraft patches for 10 of them. Those patches are applied to 1,055 (93.9%) distinct broken artifacts in an automated manner in 2,948 fixes. Among all the fixes, 44% of them are related to broken project dependencies retrieved by the build system. To prevent breakages in the future, we propose dependency caching, which caches project dependencies in the Docker container of each artifact. Dependency caching is successful on 1,700 (94.7%) artifacts with an average size increase of only 5.1%. Finally, we isolated 1,257 out of 1,700 artifacts, which are reproducible even without internet. An evaluation of the reproducibility of isolated artifacts over an 8-month period revealed that reproducibility reaches 95% or higher.

There has been previous research in fixing broken builds [24, 36, 40, 42, 56, 60]. These approaches can be applied automatically to fix broken artifacts due to dependency-related issues. They range from including additional repositories from which dependencies are retrieved, to specifically changing dependency versions. However, these repair strategies do not consider environmental factors (e.g., changing the JDK or TLS versions), which we find to be critical in the long-term reproducibility of software artifacts. Furthermore, the above neither study the nature of software breakage through time nor apply prevention strategies to ensure long-term reproducibility.

Our study on reproducibility has revealed open opportunities for the research community that are applicable to defect datasets, and software in general. Software fragility is a real problem. This is compounded by the unavailability of software dependencies and bad practices in writing non-isolated tests. We highlight this by quantifying their effects on the reproducibility of BUGSWARM. We have developed multiple approaches for repairing and preventing breakage that we hope can be learned from and adopted by other datasets, as well as software engineering in general.

The contributions of this paper are as follows:

- We present a study on the reproducibility of five state-of-the-art Java software defect datasets, and show how applied reproducibility strategies fail to prevent software breakages (Section II).
- We examine the reproducibility of BUGSWARM in a 13-month period to investigate the frequency, root causes, and fixes of software breakages (Section III).
- We develop dependency caching and artifact isolation to reduce the risk of breakage and to ensure long-term reproducibility. We successfully isolate 1,257 artifacts, which have a reproducibility of handcrafted quality while being mined automatically (Section III).

II. REPRODUCIBILITY OF JAVA DEFECT DATASETS

Software defect datasets lay an essential foundation for software engineering (SE) research. In the past decade, along with the focus of fault localization, test generation and automated program repair on Java projects, several Java defect datasets [27, 29, 30, 32, 37, 48, 53, 58] have been created by the SE community, acting as benchmarks to evaluate proposed techniques on real-world software while helping advance the state of the art. In particular, areas such as fault localization and automated program repair require *reproducible* software defects. Reproducibility in this context means that the code can be built, tests can be run, and expected failures are observed. Sudden breakage of defect artifacts, i.e., broken builds, the absence of expected failures or the occurrence of unexpected failures, can be quite problematic for the SE community as it would lead to wrong findings, nonreplicable past studies, or even to the inability to perform future studies.

This section investigates the reproducibility of five state-of-the-art Java software defects datasets: DEFECTS4J, GROWINGBUGS, BUGS.JAR, BUGSWARM and BEARS. Each artifact consists of a buggy version of the source code and the corresponding fixed version. All datasets, except for BUGS.JAR, provide scripts to build and run existing tests. Note that we only focus on datasets whose artifacts consist of whole projects. Datasets that only provide code snippets or single source files without build configurations, such as QUIXBUGS [32], are not included in this study. Also, we do not include datasets of non-functional bugs [23, 47] and web application bugs [54]. Lastly, recent datasets [27, 53] not available at the time of this study are not included. The rest of this section aims to answer the following research questions:

RQ1 What criteria and strategies do datasets use to define and ensure reproducibility?

RQ2 How are datasets affected by software breakages?

A. RQ1: Reproducibility Criteria & Strategies

Table I captures the details of the selected datasets including the number of defect artifacts included in each dataset as of writing of this paper, the number of different projects

TABLE II
CRITERIA OF REPRODUCIBILITY FOR DATASETS

Dataset	Definition of Reproducibility			
	Existence	Number	Name	Status
DEFECTS4J	✓	✓	✓	
GROWINGBUGS	✓	✓	✓	
BUGS.JAR	✓			
BUGSWARM	✓	✓	✓	✓
BEARS	✓			

these artifacts belong to, the source of the defects (issue tracker or continuous integration) and their location (source files and/or configuration files), and the kind of reference provided by the datasets (list of failing tests, build logs at time of creation, or CI historical build logs). The rest of this section focuses on investigating the reproducibility criteria and strategies employed by Java datasets to ensure reproducibility, both at creation time and long term. For this, we have examined three sources of information when available: dataset documentation, publications, and dataset infrastructure code. Below we describe our findings for each dataset.

1) DEFECTS4J*:* DEFECTS4J [6, 30] is one of the most widely-used software defect datasets, which consists of 864 artifacts manually mined over a span of 8 years. A list of reference failing tests is provided by DEFECTS4J for each artifact. These are the failing tests captured upon the *creation* of the artifact, which failed on the buggy version but passed on the fixed version. DEFECTS4J considers an artifact to be reproducible if all of its reference failing tests are observed when reproducing (running again) the buggy version, and no failing tests are observed when reproducing the fixed version. To ensure long-term reproducibility, DEFECTS4J lists the system requirements common to all artifacts, including required versions of Java, Git, SVN and Perl. Furthermore, specific dependency packages of projects are downloaded from the official web server of DEFECTS4J in the initialization stage. To check the reproducibility of DEFECTS4J, users can run a verification script provided by the maintainers.

2) GROWINGBUGS*:* GROWINGBUGS [2, 3, 29] extends DEFECTS4J's framework to *automatically* mine 570 artifacts via BUGBUILDER. Like DEFECTS4J, GROWINGBUGS provides a list of failing tests for each artifact as reference. GROWINGBUGS adopts DEFECTS4J's reproducibility criteria and relies on *some* of its reproducibility strategies. While the initialization step from DEFECTS4J is automatically run to download dependency packages from DEFECTS4J's web server, GROWINGBUGS does *not* update this server with new dependency packages used by newly added artifacts, which can result in missing packages that may need to be downloaded from third-party locations. Finally, its compatibility with DEFECTS4J allows the reuse of DEFECTS4J's verification script to examine dataset reproducibility.

3) BUGS.JAR*:* BUGS.JAR [4, 48] provides 1,158 artifacts automatically mined from bug reports of 8 large open-source projects. For each of its artifacts, BUGS.JAR provides a reference log for the buggy version, which was generated at the time the dataset was created. Also at creation time, BUGS.JAR removes tests that fail when running the fixed version based on the assumption that such tests may be unrelated to the fix. Thus, BUGS.JAR considers an artifact reproducible if there are no observed failing tests when running the curated fixed version, and if *any* failing test is observed when running the buggy version. To ensure reproducibility at creation time, each artifact is run 10 times to detect non-reproducible and flaky artifacts, which are excluded from the dataset. No additional actions are taken for long-term reproducibility. To the best of our knowledge, BUGS.JAR does not provide scripts to run artifacts and verify their reproducibility.

4) BUGSWARM*:* BUGSWARM [5, 58] is a software defect dataset that leverages Continuous Integration (CI) to automatically mine 1,941 artifacts from projects hosted in GitHub. Unlike BUGS.JAR, BUGSWARM provides reference logs for buggy and fixed versions. The reference logs are historical build logs generated by Travis-CI when the code was originally committed to GitHub and tested in the cloud. In determining reproducibility, BUGSWARM checks whether the *number* of passing & failing tests and the *names* of observed failing tests match those listed in the reference logs as well as the status of the CI build (`passing`, `failing` or `error`). No test failures are expected for the fixed version to be considered reproducible. To ensure reproducibility during creation, BUGSWARM tested the reproducibility of each artifact 5 times. The dataset includes artifacts that are reproducible more than once but less than five times, which are labeled as "flaky." We exclude all 146 flaky artifacts and consider the remaining 1,795 artifacts in this study. BUGSWARM containerizes each artifact as a Docker image in an attempt to ensure long-term reproducibility. The Docker image aims to replicate the exact environment used by Travis-CI to run the code and includes tailored scripts to run buggy and fixed versions.

5) BEARS*:* Like BUGSWARM, BEARS [1, 37] automatically mines 251 artifacts from 72 GitHub projects that use Travis-CI. However, instead of reference logs, BEARS provides for each artifact a JSON file with information collected at creation time, such as reference failing tests. Similar to BUGS.JAR, BEARS considers an artifact reproducible if there are no failing tests in the fixed version, and there is at least *one* failing test in the buggy version. During creation, each artifact is run twice; artifacts with flaky tests are excluded from the dataset. BEARS relies on Maven for the build process. No other actions are taken for long-term reproducibility. A script is available to run all artifacts, but reproducibility is not verified.

RQ1: Reproducibility criteria vary across datasets ranging from the existence of *any* failure to matching the number and/or names of failing tests, and CI status. Various reproducibility strategies are applied at creation time, the most popular of which runs artifacts multiple times to identify flakiness. Strategies for long-term reproducibility include managing dependencies and using containers.

TABLE III
REPRODUCIBILITY OF SOFTWARE DEFECT DATASETS. RESULTS FOR DEFAULT CRITERIA ARE SHOWN IN BOLD.

Dataset	# Artifacts	Reproducibility for Different Criteria			
		Existence	Number Match	Name Match	Status Match
DEFECTS4J	864	837 (96.9%)	837 (96.9%)	**837** (96.9%)	N/A
GROWINGBUGS	570	175 (30.7%)	170 (29.8%)	**170** (29.8%)	N/A
BUGS.JAR	1,158	**308** (26.6%)	303 (26.2%)	303 (26.2%)	N/A
BUGSWARM	1,795	1,392 (77.5%)	1,388 (77.3%)	1,387 (77.3%)	**1,239** (69%)
BEARS	251	**137** (54.6%)	134 (53.4%)	134 (53.4%)	N/A

B. RQ2: Software Breakages in Java Defect Datasets

In this section, we discuss how state-of-the-art datasets are affected by software breakages given the various reproducibility criteria and strategies discussed in the previous section.

1) Criteria for Reproducibility: As discussed earlier, various reproducibility criteria are used across datasets. Here we categorize these criteria as **Existence**, **Number Match**, **Name Match** and **Status Match**. **Existence** means that the artifact is reproducible if there is at least *one failing test* when running the buggy version. **Number Match** means that the artifact is reproducible if the *number* of observed failing tests (and passing tests, if required by the dataset) matches the number of tests reported by a given dataset in their reference data. **Name Match** requires the *names* of failing tests to match. In addition of fulfilling Number Match and Name Match, **Status Match** also requires *CI build status* to match. Note that all of these criteria are based on the assumption that the fixed version should never have failing tests.

Table II shows the *default* reproducibility categories per dataset. BUGS.JAR and BEARS only check Existence of failing tests. DEFECTS4J and GROWINGBUGS use Name Match, and therefore also require Number Match and Existence. BUGSWARM adopts the most strict criterion Status Match that requires everything including CI build status to match.

2) Experimental Setup: When running each artifact, we first check out the buggy version, build the project, run tests and gather results as *reproduced* logs. Then we checkout the fixed version and repeat the process of building, testing and collecting reproduced logs. Because BUGS.JAR only provides a `developer-patch.diff` instead of a fixed version, we apply the patch to the buggy version via `git apply` to generate the fixed version.

We comply with the environmental requirements of each dataset when specified. DEFECTS4J and GROWINGBUGS explicitly require the use of Java 8, and BUGSWARM includes the necessary Java versions in the Docker images of its artifacts. On the other hand, BUGS.JAR and BEARS do not specify a required Java version for their artifacts. We use Java 8 because (1) Java 8 is the earliest version that is still in life cycle at the time of conducting this study (Java 7 and earlier versions are deprecated and thus inaccessible), and (2) this study aims to report reproducibility from a user's standpoint, and therefore it is not intended to debug individual artifacts. Note that there are artifacts from BUGS.JAR and BEARS that were originally built with Java 7 or older versions. We discuss the implications of unspecified Java versions in Section IV.

We use the scripts provided by DEFECTS4J, GROWINGBUGS, BUGSWARM and BEARS to run their artifacts. For BUGS.JAR, which does not provide scripts, we leverage the reference logs to infer the commands to build and run tests. Then we verify the correctness of the commands by comparing the test classes executed with those listed in BUGS.JAR's reference logs. For all datasets, we parse the reference logs and the reproduced logs to extract test information, and in the case of BUGSWARM, Travis-CI build outcome. Then we calculate reproducibility, i.e., the percentage of artifacts in each dataset match their reference logs, and thus are still reproducible.

All datasets except for BUGSWARM explicitly exclude flaky artifacts. For BUGSWARM, we do not consider artifacts marked as "flaky." Nevertheless, we run the artifacts of each dataset three times to rule out unknown flakiness. Note that we do not observe major differences across runs, and this section presents the results for the *first* run dated in July 2022.

3) Results: Table III shows the reproducibility of each software defect dataset. Interestingly, the reproducibility under the category of Existence seems quite accurate to be representative. Matching number and/or name of tests just slightly decreases the reproducibility of all datasets. In fact, only 18 artifacts are reproducible under Existence but broken under Number Match or Name Match. We manually inspected all of them and found that 10 artifacts have missing expected failing tests when reproducing and 8 artifacts have unexpected failing tests. These artifacts still have observed failing tests but the number or name does not match the reference provided by datasets. On the other hand the category Status Match, only used by BUGSWARM, has a more significant impact on the overall reproducibility. While results for each criterion can be found in Table III, the rest of this discussion focuses on the *default criterion* used by each dataset.

DEFECTS4J achieves the highest reproducibility of 96.9%. Among 864 artifacts, only 27 are broken. Note that maintainers of DEFECTS4J have deprecated 29 artifacts due to behavioral changes introduced under Java 8. But we still examine reproducibility of all artifacts and only 27 of the 29 deprecated artifacts are broken in our run. Recall that DEFECTS4J is a handcrafted dataset that has taken more than 8 years to grow to 864 artifacts. The tedious and time-consuming human labor, among other things, has paid off in achieving a remarkable level of reproducibility.

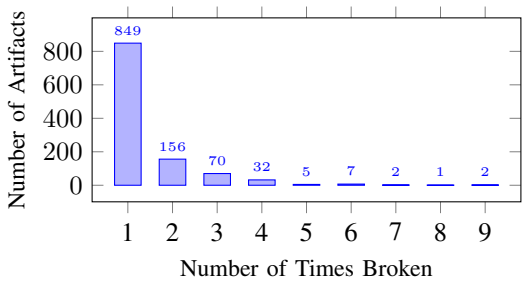

Fig. 1. Count of Artifacts with a Number of Breakages

As an extension of DEFECTS4J, GROWINGBUGS automatically mined 570 new artifacts without human intervention. However, the reproducibility of these auto-mined artifacts is considerably lower than that of the handcrafted DEFECTS4J artifacts, despite being newer. We find that 400 out of 570 artifacts are currently broken 15 months after their creation when using their default criterion Name Match, resulting in a reproducibility rate of only 29.8%. Using the least restrictive criterion Existence increases reproducibility slightly to 30.7%.

For BUGS.JAR, only 308 out of 1,158 artifacts are found to be reproducible, i.e., 26.6%. Despite running each artifact 10 times to ensure reproducibility at creation time, reproducibility of 850 artifacts faded away since the dataset creation in October 2017. Interestingly, despite that BUGS.JAR uses the most loose criterion Existence by default, 303 out of 308 reproducible artifacts remain reproducible when applying stricter reproducibility criteria. If we apply Number Match and Name Match, the reproducibility will be both 26.2%.

We find that 1,239 out of 1,795 artifacts are reproducible in BUGSWARM. The containerization strategy seems to contribute to a slightly higher reproducibility of 69% after 32 months since its creation, which is the highest among those datasets created automatically. Recall that BUGSWARM's default reproducibility criterion is the most restrictive: Status Match. Less restrictive criteria Number/Name Match and Existence lead to a higher reproducibility of 77.3% and 77.5%, respectively. The difference in reproducibility is due to mismatches of the overall status of the Travis-CI build. There are no failing tests in the fixed version, and the number and names of failing tests in the buggy version match the reference logs. However, when reproducing the fixed version, the Travis-CI build status is `failing` because some other CI steps fail after tests run. In this case, the artifact is deemed broken by the default reproducibility criterion of BUGSWARM, but reproducible for all other criteria.

Finally, the reproducibility of BEARS is 54.6% with 137 out of 251 artifacts being reproducible. After 46 months from creation, 117 artifacts have broken. Similar to BUGS.JAR, despite using the least restrictive Existence criterion by default, reproducibility remains almost the same when applying Number/Name Match, achieving a reproducibility of 53.4%.

Interestingly, while there is an overlap of projects from which some datasets mine artifacts, reproducibility at the project level varies significantly across datasets. For instance, both DEFECTS4J and GROWINGBUGS mine artifacts from `JacksonDatabind`, and both BUGSWARM and BEARS mine artifacts from `raphw/byte-buddy`. However, in DEFECTS4J, all 112 artifacts (100%) mined from `JacksonDatabind` are reproducible while in GROWINGBUGS, only 12 out of 39 artifacts (30.8%) are reproducible. Similarly, 351 out of 361 artifacts (97.2%) mined from `raphw/byte-buddy` in BUGSWARM are reproducible while none of the 5 artifacts in BEARS are reproducible.

Note that some BUGSWARM artifacts correspond to different build jobs of a same project version, where build jobs differ in the environment used to build (and test) a given project version. We grouped the 1,795 BUGSWARM artifacts by project version, which resulted in 1,152 groups from which 430 groups have more than one artifact. We found that in 67 out of 430 groups some build jobs are reproducible while others are broken. For example, `raphw-byte-buddy-202917181` and `raphw-byte-buddy-202917180` are build jobs of the same project version, but differ in reproducibility. Among the remaining 363 groups, in 262 all build jobs are reproducible while in 101 all build jobs are broken.

Our manual inspection of breakages revealed that artifacts from different datasets are broken due to similar reasons. Reasons for breakages include compile errors, unexpected test failures observed in the fixed version, and expected test failures in the buggy version not being observed. There are also common root causes leading to breakage. For example, the deprecation of Java 7 caused artifact breakages for all datasets except for BUGSWARM, which leverages containerization to use the reference environment to run artifacts. Other root causes include missing compatible dependencies and deprecated insecure links. Section III will provide an in-depth investigation in the context of BUGSWARM.

RQ2: All datasets experience breakages, especially those created automatically. DEFECTS4J, the only handcrafted dataset, reaches the highest reproducibility of 96.9%, while BUGS.JAR has the lowest 26.6%. Despite adopting different criteria, reproducibility for a given dataset only varies slightly. On the other hand, artifacts from a same project can have different reproducibility across datasets.

III. A CASE STUDY ON REPRODUCIBILITY

The goal of this study is to determine how software defect artifacts experience breakage through time, and how to ensure long-term reproducibility. Our study is threefold. First, we study the frequency at which software artifacts break. Second, we examine breakages to identify common root causes and fixes. Lastly, we discuss and evaluate strategies to prevent software breakage in the context of software defect datasets. We aim to answer the following research questions:

RQ3 How often do software breakages occur?
RQ4 What are root causes & fixes for software breakages?
RQ5 How can we prevent software breakages?

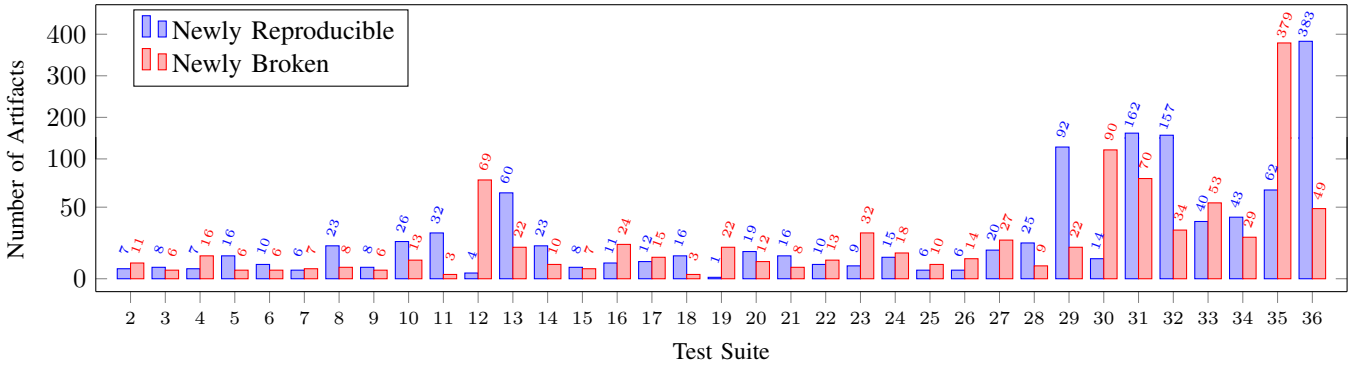

Fig. 2. Number of Newly Reproducible and Newly Broken Artifacts for Each Test Suite

BUGSWARM *Dataset.* This case study is conducted on the dataset BUGSWARM [5, 58]. BUGSWARM is a software defect dataset designed to continuously grow. Upon our first adoption of BUGSWARM, the dataset included 1,941 Java artifacts from 120 different open-source projects hosted in GitHub that use Travis-CI, from which 146 are marked as "flaky." BUGSWARM provides a Docker image for each artifact, which consists of the buggy and fixed versions of the source code, the exact build environment in which the code was first built, and a script to build the project and run existing tests. All the artifacts in BUGSWARM use a build system. Among non-flaky artifacts, there are 1,640 artifacts that use the Maven build system, 69 that use Ant, and 86 that use Gradle.

We select BUGSWARM as the subject dataset for this study because: (1) it is an automatically created software defect dataset subject to software breakage, (2) its artifact containerization facilitates investigating and fixing breakages, (3) it provides software defects both on source code and configuration files, (4) its scale is relatively larger than others, and (5) it is in active development.

Experimental Setup. We study all 1,795 non-flaky Java defect artifacts from BUGSWARM, all of which were reproducible when the dataset was created. At regular intervals, on average every 11.7 days over a 13-month period, we tested the reproducibility of the software artifacts in BUGSWARM. Each run of reproducibility test for all artifacts is referred as a *test suite*. We collected the results for a total of 36 test suites. The testing process consisted of (1) pulling the corresponding Docker images, (2) running the build scripts for both buggy and fixed versions, which download dependencies, build the projects, and run tests, and (3) comparing the reproduced logs against the reference logs, which are the Travis-CI historical build logs. We report reproducibility results as calculated by the criteria originally used by BUGSWARM—the status of the build (`passing`, `failing`, or `error`) is compared along with the number of passing & failing tests and the names of failing tests (if any), for both buggy and fixed versions. An artifact is deemed *reproducible* if all of these attributes match, otherwise the artifact is *broken*. Recall that this is the the most strict reproducibility criterion discussed in Section II-B.

A. RQ3: Breakage Frequency

In this section we investigate how often software breakages occur. In particular, we first find how many of all artifacts have ever suffered from breakage during the 13-month study, and whether this is a one-time occurrence. Second, we investigate the overall frequency of breakage, e.g., whether artifacts gradually break on every test run, or there are specific points in time in which large batches of artifacts break at once.

To understand the scale of breakage, we count the number of artifacts with breakages across all 36 test suites. The breakage count increments every time the status of an artifact changes from reproducible to broken. We find that 1,124 out of 1,795 artifacts (62.6%) broke at least once during the 13-month study. These artifacts belong to 86 out of 120 Java projects (71.6%). This shows that breakage is not a rare occurrence and that it affects a large majority of artifacts across different projects. Figure 1 shows the distribution of artifacts in terms of the number of breakages reported during the study. A total of 275 (15.3%) artifacts broke multiple times with almost half of those breaking three or more times. Note that the number of artifacts decreases exponentially as the number of breakages increases. This is expected as broken artifacts remain broken until a patch is applied to fix the breakage.

On average, 430 (24%) artifacts remained broken in each test suite. All artifacts were reproducible when created, but as time passed they suffered from breakages for various reasons, which we describe later in Section III-B. Figure 2 shows the number of *newly reproducible* and *newly broken* artifacts identified in various test suites. A newly reproducible artifact in a given test suite is an artifact that was broken in the previous test suite, but that just became reproducible again because a patch was applied to fix the breakage. A newly broken artifact is an artifact that was reproducible in the previous test suite, but it is now broken. For example, in test suite #36 in Figure 2, there are 383 previously broken artifacts that are fixed, but an additional 49 artifacts are now broken.

For the 36 test suites, the average of newly reproducible artifacts is 38, and 32 for newly broken artifacts. This suggests that the number of artifact breakages in the context of two consecutive test runs is relatively small, considering a total

TABLE IV
ROOT CAUSES AND PATCHES FOR BREAKAGES

Category #	Root Cause	Patch	# Patches Applied	Percentage
1	Maven TLS Failure	Update TLSv1.0 to TLSv1.2	730	24.8%
2	Unavailable PPAs	Remove PPAs no longer available	679	23%
3	Unavailable Ubuntu Release	Change URLs for repository	392	13.3%
4	Insecure Link	Change URLs using HTTP to HTTPS	339	11.5%
5	Unavailable JDK Version	Retrieve JDK version from official repository	316	10.7%
6	Unavailable Gradle Plugin	Update URL of specific Gradle Plugin	158	5.4%
7	Unavailable NodeJS Installer	Change URL to retrieve NodeJS installer	91	3.1%
8	Incompatible NPM Package	Pin NPM package version	91	3.1%
9	Unavailable XML	Update URL to retrieve DTD files	83	2.8%
10	Deprecated Checkstyle Link	Replace deprecated checkstyle URL	69	2.3%
11	Unexpected Test Failures	N/A	-	-

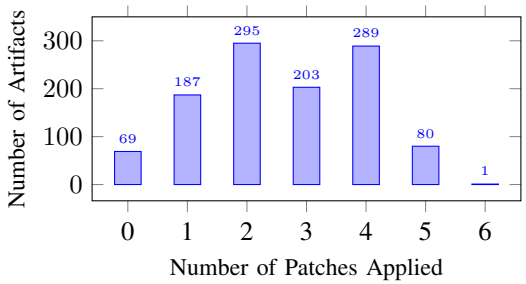

Fig. 3. Count of Artifacts with a Number of Patches

of 1,795 artifacts. Nevertheless, this is problematic as it demonstrates a need for high effort to maintain reproducibility; there are 32 new artifacts that experience breakage every 11.7 days on average. The maximum number of newly broken artifacts across all test suites is 379, and the maximum number of newly reproducible artifacts across all test suites is 383. The median for newly broken artifacts is 14, and for newly reproducible artifacts is 16. The above demonstrates the brittleness of artifact reproducibility and the challenge facing the maintenance of long-term reproducibility for an automatically created dataset.

RQ3: A total of 1,124 (62.6%) unique artifacts broke during our 13-month study. This includes artifacts from 86 (71.6%) different Java projects. For each test suite, 32 newly broken artifacts were observed on average. The rate at which artifacts break imposes a great challenge on the long-term maintainability of the dataset.

B. RQ4: Breakage Root Causes and Fixes

Our 13-month study revealed a total of 1,606 individual instances of software breakage that affected 1,124 unique artifacts. We manually examined each breakage to identify root causes and potential fixes. Specifically, for each instance, we inspected the errors in the reproduced logs, and also referred to the configuration and source files when necessary. To minimize bias, three individuals worked independently to inspect each breakage instance to generate a consensus on its root cause.

Our manual investigation uncovered 11 root causes of breakage. Table IV provides a brief description of the root causes and their patches. The patches can be categorized into: (1) system dependency and (2) project dependency. A system dependency patch fixes issues of dependency installation via APT. APT is a package manager for Linux that allows users to install, update, and remove packages retrieved from official package repositories such as Ubuntu Archive or from unofficial Personal Package Archives (PPAs). A project dependency patch modifies dependencies specified in build configurations. An important requirement for patches applied to BUGSWARM is that they do *not* make significant changes to the artifact. These patches are the reason for the newly reproducible artifacts shown in Figure 2. Note that all patches are manually created but applied in an automated manner. Among 1,124 artifacts that break at least once, we applied 2,948 patches on 1,055 (93.9%) of them. Note that 868 out of 1,055 (82.3%) fixed artifacts required multiple patches. Figure 3 shows the distribution of artifacts based on the number of patches applied. The rest of this section discusses each root cause and fix strategy.

a) Maven TLS Failure: This is the most common among all root causes, affecting 730 artifacts from 68 distinct projects. This issue is caused by the deprecation of TLSv1.0. The patch updates TLSv1.0 to TLSv1.2 and falls into the category of project dependency patch. This patch accounts for 24.8% of all the applied fixes.

b) Unavailable PPAs: A total of 679 artifacts from 56 different projects were affected by unavailable PPAs. The breakage occurs when a previously accessible resource becomes unavailable. These particular resources are PPAs, i.e., Personal Package Archives, which are used on Ubuntu to package system dependencies. However, we found that these dependencies were actually not necessary for the artifacts to build successfully, and thus it is safe to remove the installation command that was causing the breakages. More specifically, the patch consists of removing the PPAs from the list of repositories that are retrieved when running `apt-get update`. This is considered as a system dependency patch, which corresponds to 23% of the fixes.

c) Unavailable Ubuntu Release: Ubuntu 12.04 LTS's Precise Pangolin packages and releases were migrated to a

new repository, making previous URLs used by the package managers to become stale. The patch consists of updating the APT's sources list to install the Ubuntu 12.04 LTS releases via the new URL. This affected 392 artifacts from 7 distinct projects and accounted for 13.3% of the fixes.

d) Insecure Link: Maven central repository stopped the transfer of dependencies for HTTP protocol requests, instead requiring HTTPS. The transfer failed with a `501` return code resulting in unresolved dependencies. This affected 339 artifacts from 44 distinct projects. The fix was to replace URLs HTTP with HTTPS in affected URLs, and it accounted for 11.5% of the fixes.

e) Unavailable JDK Version: This breakage occurred when artifacts relied on a third-party server to retrieve a specific version of the Java Development Kit (JDK), however the server was no longer accessible. The patch simply retrieves the missing JDK package from the official repositories. This patch is applied to 316 artifacts. Note that while the number of artifacts is large, they belonged to only 3 projects. Overall, this category represents 10.7% of the fixes.

f) Unavailable Gradle Plugin, XML and NodeJS Installer: Unavailable Gradle Plugin is observed when plugins originally hosted in non-central repositories are no longer accessible. Similarly, Unavailable XML occurs when `.xml` files become inaccessible due to URL changes. Finally, Unavailable NodeJS Installer also refers to a stale URL. The installer was previously downloaded from an Amazon Simple Storage Server (S3) Bucket instead of the official NodeJS installer source URL. All these examples reveal the danger of relying on third-party sources to host required resources. All together, these accounted for 332 fixes on artifacts from 3 distinct projects, and represented 11.3% of all applied fixes.

g) Incompatible NPM Package: This is caused by an incompatibility with the latest release of the NPM package. Our investigation found that NPM's semantic versioning was set to be greater than or equal to a certain version, e.g., `"my_dep" : ">1.0.0"`. As a result, a latest version was installed instead of the original version used by the project. The patch consists of pinning the specific package version used when the project was originally built. This affected 91 artifacts all from one project, accounting for 3.1% of the breakage fixes.

h) Deprecated Checkstyle Link: These breakages occur when the link to retrieve the Checkstyle package is deprecated. Checkstyle is a development tool to examine the adherence of coding standards. The deprecation affected 69 artifacts from a project that retrieves the Checkstyle package during the build process. The patch simply updates the link of the Checkstyle package, accounting for 2.3% of the fixes applied.

i) Unexpected Test Failures: Finally, 69 artifacts were broken due to unexpected test failures; the number of failing tests do not match the reference log. Our further investigation identified four sources of test failures: expired API key for tests, bad SSL certificate, timeout tests, and errors from the Gradle build system, for which we do not provide patches.

Final Remarks for Root Causes and Fixes: Artifacts break for different reasons, and often experience multiple breakages over time. Our patches are applied to 1,055 artifacts in 2,948 fixes. Note that *Maven TLS Failure*, *Insecure Link*, *Unavailable Gradle Plugin* and *Deprecated Checkstyle Link* are patches that fix project dependencies, which are retrieved by the build system and account for 44% of all fixes. The rest (except for unexpected test failures) could be attributed to system dependencies. We discuss next how project dependencies represent low-hanging fruit for preventing future breakage.

> **RQ4:** We identify 11 root causes of breakages and manually crafted patches to fix ten of them. We automatically apply 2,948 patches on 1,055 artifacts. 1,296 (44%) of the fixes are related to project dependencies.

C. RQ5: Breakage Prevention

In this section, we discuss preventive strategies to ensure long-term reproducibility of software defect artifacts. First, we describe how we leverage popular build systems to cache project dependencies. This was inspired by the insight from our study on root causes and fixes (Section III-B): 1,296 of all patches are applied to fix errors from dependencies retrieved by build systems. We discuss the impact of dependency caching on the reproducibility of 20 additional test suites over a time period of 8 months. Finally, we discuss the need for full isolation of artifacts to ensure long-term reproducibility.

1) Dependency Caching: To prevent BUGSWARM artifacts from breaking due to project dependencies managed by their build system, we add required dependencies for each artifact to the corresponding Docker image (recall that artifacts from BUGSWARM are instantiated as Docker containers). We refer to this process as *dependency caching*, which can be automated with the help of the build systems (e.g., Maven). The ultimate goal is to make artifacts independent from external repositories. Artifacts whose dependencies have been cached (also referred to as *cached artifacts*) run in the same way as the original artifacts, but no longer need access to an external central repository to download dependencies.

a) Approach for Dependency Caching: We use the local repository configuration provided by the build system. We describe our process for Maven, which is similar for Gradle and Ant. We set a local repository path in Maven's XML configuration file. This local repository is then used to automatically save dependency files downloaded from remote repositories when the build script invokes `mvn install` [7]. After downloading all dependencies, we configure the artifact to build offline with the `-o` option in `mvn install`. This option forces the project to build only from dependencies in the local repository. Note that Ant supports caching but does not have an offline mode. So we assume that the cached Ant artifacts are cached correctly as long as their build script does not break. We check for the existence of additional dependencies that may require caching in Section III-C2.

b) Effectiveness of Dependency Caching: A total of 1,700 artifacts in BUGSWARM are reproducible after the patching process described in Section III-B, and thus eligible

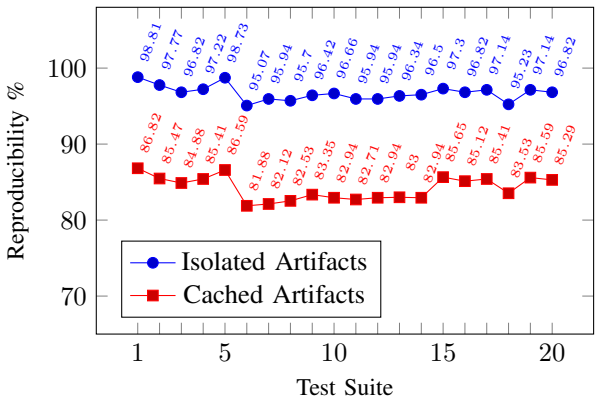

Fig. 4. Reproducibility of Cached and Isolated Artifacts

for dependency caching, including 1,566 Maven artifacts, 71 Gradle artifacts, and 63 Ant artifacts. We successfully cached all artifacts automatically. Cached artifacts are tested by running their build script using the build system's offline mode to restrict their network access. Since dependency caching adds new files to the Docker container, the size of Docker images in BUGSWARM increases. However, to our surprise, the increase in size is rather small. For each artifact, after caching all its required dependencies, the size increase rate is 5.1% on average, and the 25th, 50th, 75th, and 90th percentiles of the size increase are 2.5%, 3.5%, 5.9%, and 12.2%, respectively.

We evaluated the reproducibility of cached artifacts across 20 test suites over a period of 8 months, subsequent to our initial study (Section III-A), and a few weeks after the caching process concluded. The results are shown in Figure 4. While we observe a decrease in reproducibility, the reproducibility is mostly maintained in the 81% to 85% range with no maintenance effort. The reason for breakage is mainly due to dependencies that were not captured by our approach. Recall that we cache dependencies via Java build systems like Maven, but there are some artifacts fetching dependencies from other sources. Unexpected tests failures are the main reason for the variability across test suites. Some tests fail due to the temporary unavailability of remote services. For example, tests of `apache-dubbo-416755517` invoke the RPC service, which was unavailable during the runs of test suites #6 & #7 and restored for later test suites.

2) Artifact Isolation: Ideally, a cached artifact should be reproducible even without internet access because all its required dependencies are available in its Docker container. If an artifact is reproducible without access to external resources like the internet, then it should be reproducible in perpetuity. To verify if cached artifacts are completely reproducible, we tested the reproducibility of cached artifacts when disconnected from the internet. We found that only 920 out of 1,700 (54.1%) artifacts were reproducible.

We manually inspected the remaining 780 artifacts to identify the root cause. We identified three root causes that we were able to fix, bringing the number of successfully isolated artifacts to 1,257 (73.9%). The most frequent issue was build failure during the Travis initialization stage. This was observed for artifacts that use Travis's temporary caching functionality, which uploads specific dependencies to the internet for later use [9], and thus failed in isolation. This initially affected 243 artifacts that are now reproducible in isolation after disabling Travis caching. The second reason was related to downloading packages via NPM from the internet. Note that our caching process only caches dependencies via build systems. Thus, without internet access, the build script is not able to retrieve NPM packages and thus fails. This affected 88 artifacts that are now reproducible. The remaining 6 artifacts had port issues that we fixed by enabling localhost within Docker containers.

TABLE V
REPRODUCIBILITY WITHOUT INTERNET ACCESS

Description	Number	Ratio
Cached and reproducible artifacts	1,700	100.00%
Artifacts currently reproducible offline	1,257	73.9%
Not Reproducible – Internet Access		
Cached plugin accesses the internet	157	9.2%
Internet access in unit tests	80	4.7%
Build script accesses GitHub at runtime	68	4%
Not Reproducible – Caching Issues		
Incompletely cached, missing dependencies	51	3%
Irregular dependency installation	32	1.9%
Misc. runtime errors	55	3.2%

A total of 443 (26.1%) cached artifacts are broken without internet. Table V shows two main categories: (1) *Internet Access*, artifacts that explicitly access the internet during the build process, and (2) *Caching Issues*, artifacts that access the internet to upload or download files in ways that are not captured by our dependency caching techniques.

Internet Access accounts for 17.9% of cached artifacts. It is unusual to observe software artifacts that download content from hardcoded URLs at runtime because this practice creates test instability. Among these artifacts, calls to the internet were included in the tests, build scripts, or in the installation process of cached dependencies. These are not recommended developer practices for offline mode artifacts according to Maven's documentation [7]. These artifacts would be much more difficult to reproduce in isolation, and each fix would be artifact specific, so we leave the process of mocking specific remote resources for future work. *Caching Issues* affect 8.1% of the artifacts. This is mainly due limitations of our caching technique. For example, 51 Ant and Gradle cached artifacts are missing system dependencies. Unlike the artifacts that explicitly make calls to the internet, these artifacts are likely to be reproducible with refinements to the dependency caching approach or manual caching of missing files.

Like cached artifacts, we evaluated reproducibility of isolated artifacts with 20 test suites run over 8 months. Figure 4 shows the reproducibility of isolated artifacts along with that the reproducibility of artifacts cached. Note that reproducibility of isolated artifacts stays above 95%. We inspected a few of the broken artifacts, and found that breakage is due to a timeout. In particular, this only affects artifacts

belonging to one of two projects: `rinde-RinSim` and `spring-projects-spring-data-jpa`. Further investigation is required to determine the reasons for the timeout.

3) Final Remarks: Note that BUGSWARM as of July 2022 included all the patches from Section III-B, which were acknowledged and pushed to BUGSWARM's main Docker repository. However, despite all the breakage fixes, reproducibility dropped to 69% due to further breakage. On the other hand, the official version of the dataset does *not* include yet our mechanisms for breakage prevention. We applied these mechanisms to a copy of the dataset, which allows us to further evaluate the impact of dependency caching and artifact isolation. We demonstrate that breakage prevention contributes to maintaining high reproducibility after 8 months of 85.29% and 96.82% for cached and isolated artifacts, respectively, in comparison to 69% reported when no breakage prevention is in place. This is remarkable as the reproducibility is on par with that of handcrafted DEFECTS4J. Furthermore, while there is a raising concern that limitations on size and diversity of current defect datasets might introduce bias, the automated process of caching dependencies and isolating artifacts enables the continuous growth of a large-scale and diverse dataset with long-term reproducibility.

RQ5: Dependency caching is effective in preventing breakages. Isolation enables reproducibility even without internet. Reproducibility of 1,257 isolated artifacts stays above 95%, which is on par with handcrafted quality.

IV. THREATS TO VALIDITY

Our study on five Java datasets may not generalize to other languages, or even other Java datasets. However, we believe that our choice of language and datasets is representative; Java is the most popular language in fault localization and automated program repair, and we consider *all* available datasets that provide complete Java projects, which is a requirement to build and run the code.

The lack of runtime environment information for some datasets is another potential threat. In particular, BUGS.JAR and BEARS do not specify the Java version required to reproduce their artifacts, which increases the risk of incorrect execution. We contacted authors of BUGS.JAR and BEARS to request information about the runtime environment of their dataset, but we did not receive a response. Thus, to reduce this risk, we manually examined all artifacts in these datasets and identified 147 out of 1,158 artifacts from BUGS.JAR and 95 out of 251 artifacts from BEARS that require Java 7 or older. To measure the impact of this in our study, we set up a virtual machine with deprecated Java versions and ran the artifacts. Despite using the required deprecated Java versions, all artifacts were still broken. The reason was the Maven TLS Failure discussed in Section III-B. Finally, several artifacts across datasets belong to projects from the Apache Software Foundation, which may impose IP bans. We have ensured breakages do not occur because of this reason.

Despite all manual efforts, there is still room for human error when inspecting breakages. We minimize this risk by having multiple people do manual inspections following a uniform procedure. While the root causes of breakages are observed in all five datasets, the fixes we provide are specific to BUGSWARM's breakages. It remains future work to ensure such fixes generalize to other datasets. Additionally, although all the fixes are applied in an automated manner, the process of creating patches remains manual. However, the number of broken artifacts will significantly decrease as new artifacts are successfully cached and isolated. Finally, our dependency caching approach heavily relies on the build systems. It remains future work to adapt this approach beyond the build systems used by BUGSWARM's artifacts.

V. RELATED WORK

a) Understanding Build Failures: Prior studies have analyzed build failures based on 26.6 million builds at Google [49], 3412 builds from a commercial web application [31], and 2.6 million CI builds [11], finding that common build errors range from missing dependencies and other resources to test failures. Others have studied the *feasibility* of building software, such as the top 200 Java projects from GitHub [25], multi-language software packages [43], broken snapshots of Java Maven projects [59], and 7200 Java projects that use different build systems [55]. Unlike the above, we study builds that were reproducible *after* their commit time and track their reproducibility to present day. We find that the main reason for breakage is related to dependencies, which is touched on by [25, 31, 49, 59].

b) Fixing Broken Builds: There are techniques that automatically fix broken builds. BuildMedic [36] fixes dependency-related broken Maven builds by applying simple fix strategies such as Version Update or Dependency Deletion, which effectively fix 54% of the broken builds in 23 open-source projects. HireBuild [24] is a pattern-based patch generation approach for Gradle build scripts that fixes 45% of previously unseen broken builds. PyDFix [42] detects and then automatically fixes unreproducible builds due to dependency conflicts for Python projects. Vassallo et al. [60] presents BART, which provides build fix suggestions from StackOverflow. Deep learning approaches have also been introduced to fix compilation errors [40], to predict edits to fix a broken AST [56], and to fix broken Dockerfiles [26]. Similar to us, [24, 36, 60] use either common fix strategies or patterns that are empirically discovered. Furthermore, the fix for a broken build can be performed in the source files [40, 56, 60], or in the build files [24, 36, 42]. As the artifacts in our study were previously reproducible, the issue is likely related to build files. While BuildMedic and HireBuild can repair Java build-related issues, each is specific to one build system, while the builds in our study encompass three build systems.

c) Studying Reproducibility: Different aspects of software reproducibility have been studied over the years, which include quality issues and reproducibility of Dockerfiles [14], reproducibility of 30 million CPAN (Comprehensive Perl

Archive Networks) builds between 2011 and 2016 [68], reproducibility of past snapshots [38], reproducibility of GITHUB ACTIONS workflow runs [67], and reproducibility of specific kinds of bugs such as performance bugs [23], error propagation bugs [15], and concurrency bugs [62]. Furthermore, Frattini et al. [16] provide a bug taxonomy from the perspective of reproducibility with two categories: workload-dependent and environment-dependent. 18% of bug reports from MySQL Server are environment-dependent. Cavezza et al. [12] investigate environment-dependent failures reported for MySQL Server, and how altering the environment affects the reproducibility of those failures. There are also frameworks to *prevent* breakage by preserving software and its dependencies from source to execution [45, 57]. In this paper we investigate the impact of software breakages over time for general software defects, propose fix strategies for breakage and discuss preventive measures to ensure long-term reproducibility.

VI. CONCLUSIONS

Software defect artifacts are not immune to breakages. We conducted a study on the reproducibility of five state-of-the-art Java software defect datasets. Empirical results show all datasets suffer from breakages despite multiple strategies applied to ensure reproducibility. We then present a case study on the reproducibility of BUGSWARM, which was performed on 36 test suites in a 13-month period. We find that 62.6% of the artifacts break at least once during study. We manually identified 11 root causes and handcrafted 10 patches to fix breakage. We automatically applied 2,948 fixes on 1,055 artifacts to reestablish reproducibility. Furthermore, we proposed dependency caching to effectively prevent breakages and isolated 1,257 artifacts that are reproducible even without internet connection. The reproducibility of isolated artifacts stays above 95%, which is on par with handcrafted DEFECTS4J. Our study on reproducibility revealed open opportunities on the design of software defect datasets. The full replication package of this study is publicly accessible at https://github.com/ucd-plse/on-the-reproducibility.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation award CNS-2016735. We would like to thank Pengcheng Ding, Alex Dunn, Robert Furth, Ryan Jae, Eric Li, David Tomassi, Erin Winter, and Tony Xiao for their help setting up the testing infrastructure for BUGSWARM and for their work inspecting, classifying and fixing artifact breakages.

REFERENCES

[1] bears-bugs/bears-benchmark: An Extensible Java Bug Benchmark for Automatic Program Repair Studies, 2022. URL https://github.com/bears-bugs/bears-benchmark.

[2] liuhuigmail/GrowingBugRepository: A bug repository that keeps growing, 2022. URL https://github.com/liuhuigmail/GrowingBugRepository.

[3] jiangyanjie/BugBuilder: Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems, 2022. URL https://github.com/jiangyanjie/BugBuilder.

[4] bugs-dot-jar/bugs-dot-jar: Bugs.jar: A Large-scale, Diverse Dataset of Bugs for Java Program Repair, 2022. URL https://github.com/bugs-dot-jar/bugs-dot-jar.

[5] BugSwarm/bugswarm, 2022. URL https://github.com/BugSwarm/bugswarm.

[6] rjust/defects4j: A Database of Real Faults and an Experimental Infrastructure to Enable Controlled Experiments in Software Engineering Research, 2022. URL https://github.com/rjust/defects4j.

[7] Maven – Introduction to Repositories. https://maven.apache.org/guides/introduction/introduction-to-repositories.html, 2022.

[8] Artifact Review and Badging - Current. https://www.acm.org/publications/policies/artifact-review-and-badging-current, 2022.

[9] Travis Caching. https://docs.travis-ci.com/user/caching/, 2022.

[10] M. Baker. Reproducibility crisis. *nature*, 533(26):353–66, 2016.

[11] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *MSR*, pages 356–367. IEEE Computer Society, 2017.

[12] D. G. Cavezza, R. Pietrantuono, J. Alonso, S. Russo, and K. S. Trivedi. Reproducibility of environment-dependent software failures: An experience report. In *ISSRE*, pages 267–276. IEEE Computer Society, 2014.

[13] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. Begbunch: benchmarking for c bug detection tools. In *DEFECTS*, pages 16–20. ACM, 2009.

[14] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the docker container ecosystem on github. In *MSR*, pages 323–333. IEEE Computer Society, 2017.

[15] D. DeFreez, A. Bhowmick, I. Laguna, and C. Rubio-González. Detecting and reproducing error-code propagation bugs in MPI implementations. In *PPoPP*, pages 187–201. ACM, 2020.

[16] F. Frattini, R. Pietrantuono, and S. Russo. *Reproducibility of Software Bugs*, pages 551–565. Springer International Publishing, Cham, 2016.

[17] G. Gay. Challenges in using search-based test generation to identify real faults in mockito. In *SSBSE*, volume 9962 of *Lecture Notes in Computer Science*, pages 231–237, 2016.

[18] G. Gay. Generating effective test suites by combining coverage criteria. In *SSBSE*, volume 10452 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2017.

[19] G. Gay. Detecting real faults in the gson library through search-based unit test generation. In *SSBSE*, volume 11036 of *Lecture Notes in Computer Science*, pages 385–391. Springer, 2018.

[20] A. Ghanbari, S. Benton, and L. Zhang. Practical program repair via bytecode mutation. In *ISSTA*, pages 19–30. ACM, 2019.

[21] C. L. Goues, N. J. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, 41(12):1236–1256, 2015.

[22] A. Habib and M. Pradel. How many of all bugs do we find? a study of static bug detectors. In *ASE*, pages 317–328. ACM, 2018.

[23] X. Han, D. Carroll, and T. Yu. Reproducing performance bug reports in server applications: The researchers' experiences. *J. Syst. Softw.*, 156:268–282, 2019.

[24] F. Hassan and X. Wang. Hirebuild: an automatic approach to history-driven repair of build scripts. In *ICSE*, pages 1078–1089. ACM, 2018.

[25] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *ESEM*, pages 38–47. IEEE Computer Society, 2017.

[26] J. Henkel, D. Silva, L. Teixeira, M. d'Amorim, and T. W. Reps. Shipwright: A human-in-the-loop system for dockerfile repair. In *ICSE*, pages 1148–1160. IEEE, 2021.

[27] S. Herbold et al. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Softw. Engg.*, 27(6), 2022.

[28] J. Holmes, I. Ahmed, C. Brindescu, R. Gopinath, H. Zhang, and A. Groce. Using relative lines of code to guide automated test generation for python. *ACM Trans. Softw. Eng. Methodol.*, 29 (4):28:1–28:38, 2020.

[29] Y. Jiang, H. Liu, N. Niu, L. Zhang, and Y. Hu. Extracting concise bug-fixing patches from human-written patches in version control systems. In *ICSE*, pages 686–698. IEEE, 2021.

[30] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, pages 437–440. ACM, 2014.

[31] N. Kerzazi, F. Khomh, and B. Adams. Why do automated builds break? an empirical study. In *ICSME*, pages 41–50. IEEE Computer Society, 2014.

[32] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *SPLASH (Companion Volume)*, pages 55–56. ACM, 2017.

[33] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, 2005.

[34] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *ICSE*, pages 535–546. ACM, 2016.

[35] Q. Luo, K. Moran, D. Poshyvanyk, and M. D. Penta. Assessing test case prioritization on real faults and mutants. In *ICSME*, pages 240–251. IEEE Computer Society, 2018.

[36] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *SANER*, pages 106–117. IEEE Computer Society, 2018.

[37] F. Madeiral, S. Urli, M. de Almeida Maia, and M. Monperrus. BEARS: an extensible java bug benchmark for automatic program repair studies. In *SANER*, pages 468–478. IEEE, 2019.

[38] M. Maes-Bermejo, M. Gallego, F. Gortázar, G. Robles, and J. M. González-Barahona. Revisiting the building of past snapshots - a replication and reproduction study. *Empir. Softw. Eng.*, 27(3):65, 2022.

[39] C. Mandrioli and M. Maggio. Testing self-adaptive software with probabilistic guarantees on performance metrics. In *ESEC/SIGSOFT FSE*, pages 1002–1014. ACM, 2020.

[40] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian. Deepdelta: learning to repair compilation errors. In *ESEC/SIGSOFT FSE*, pages 925–936. ACM, 2019.

[41] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. FAST approaches to scalable similarity-based test case prioritization. In *ICSE*, pages 222–232. ACM, 2018.

[42] S. Mukherjee, A. Almanza, and C. Rubio-González. Fixing dependency errors for Python build reproducibility. In *ISSTA*, pages 439–451. ACM, 2021.

[43] A. Neitsch, K. Wong, and M. W. Godfrey. Build system issues in multilanguage software. In *ICSM*, pages 140–149. IEEE Computer Society, 2012.

[44] T. B. Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *ISSRE*, pages 58–68. IEEE Computer Society, 2015.

[45] L. Oliveira, D. Wilkinson, D. Mossé, and B. Childers. Supporting long-term reproducible software execution. P-RECS'18. ACM, 2018.

[46] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *ICSE*, pages 609–620. IEEE / ACM, 2017.

[47] A. Radu and S. Nadi. A dataset of non-functional bugs. In *MSR*, pages 399–403. IEEE / ACM, 2019.

[48] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *MSR*, pages 10–13. ACM, 2018.

[49] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge. Programmers' build errors: a case study (at google). In *ICSE*, pages 724–734. ACM, 2014.

[50] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T). In *ASE*, pages 201–211. IEEE Computer Society, 2015.

[51] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *ESEC/SIGSOFT FSE*, pages 237–247. ACM, 2015.

[52] J. Sohn and S. Yoo. FLUCCS: using code and change metrics to improve fault localization. In *ISSTA*, pages 273–283. ACM, 2017.

[53] X. Song, Y. Lin, S. H. Ng, Y. Wu, X. Peng, J. S. Dong, and H. Mei. Regminer: towards constructing a large regression dataset from code evolution history. In *ISSTA*, pages 314–326. ACM, 2022.

[54] Ó. Soto-Sánchez, M. Maes-Bermejo, M. Gallego, and F. Gortázar. A dataset of regressions in web applications detected by end-to-end tests. *Softw. Qual. J.*, 30(2):425–454, 2022.

[55] M. Sulír and J. Porubän. A quantitative study of java software buildability. In *PLATEAU@SPLASH*, pages 17–25. ACM, 2016.

[56] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P. Manzagol, C. Sutton, and E. Aftandilian. Learning to fix build errors with graph2diff neural networks. In *ICSE (Workshops)*, pages 19–20. ACM, 2020.

[57] D. H. T. That, G. Fils, Z. Yuan, and T. Malik. Sciunits: Reusable research objects. In *eScience*, pages 374–383. IEEE Computer Society, 2017.

[58] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In *ICSE*, pages 339–349. IEEE / ACM, 2019.

[59] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *J. Softw. Evol. Process.*, 29(4), 2017.

[60] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall. Un-break my build: assisting developers with build repair hints. In *ICPC*, pages 41–51. ACM, 2018.

[61] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung. Context-aware patch generation for better automated program repair. In *ICSE*, pages 1–11. ACM, 2018.

[62] S. Wu, K. Qiu, and Z. Zheng. An Empirical Study on Environmental Factors for Reproducing Concurrent Software Failures. *Proc. - Annu. Reliab. Maintainab. Symp.*, 2021.

[63] Q. Xin and S. P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*, pages 226–236. ACM, 2017.

[64] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *ASE*, pages 660–670. IEEE Computer Society, 2017.

[65] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE*, pages 416–426. IEEE / ACM, 2017.

[66] M. Zhang, X. Li, L. Zhang, and S. Khurshid. Boosting spectrum-based fault localization using pagerank. In *ISSTA*, pages 261–272. ACM, 2017.

[67] H.-N. Zhu, K. Z. Guan, R. M. Furth, and C. Rubio-González. ActionsRemaker: Reproducing GitHub Actions. In *ICSE-Companion*. IEEE, 2023.

[68] M. Zolfagharinia, B. Adams, and Y. Guéhéneuc. Do not trust build results at face value: an empirical study of 30 million CPAN builds. In *MSR*, pages 312–322. IEEE Computer Society, 2017.