BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes

David A. Tomassi†, Naji Dmeiri†, Yichen Wang†, Antara Bhowmick†
Yen-Chuan Liu†, Premkumar T. Devanbu†, Bogdan Vasilescu‡, Cindy Rubio-González†
†University of California, Davis {datomassi, nddmeiri, eycwang, abhowmick, yclliu, ptdevanbu, crubio}@ucdavis.edu
‡Carnegie Mellon University vasilescu@cmu.edu

***Abstract*—Fault-detection, localization, and repair methods are vital to software quality; but it is difficult to evaluate their generality, applicability, and current effectiveness. Large, diverse, realistic datasets of durably-reproducible faults and fixes are vital to good experimental evaluation of approaches to software quality, but they are difficult and expensive to assemble and keep current. Modern continuous-integration (CI) approaches, like TRAVIS-CI, which are widely used, fully configurable, and executed within custom-built containers, promise a path toward much larger defect datasets. If we can identify and archive failing and subsequent passing runs, the containers will provide a substantial assurance of durable future reproducibility of build and test. Several obstacles, however, must be overcome to make this a practical reality. We describe BUGSWARM, a toolset that navigates these obstacles to enable the creation of a *scalable, diverse, realistic, continuously growing* set of durably reproducible failing and passing versions of real-world, open-source systems. The BUGSWARM toolkit has already gathered 3,091 fail-pass pairs, in Java and Python, all packaged within fully reproducible containers. Furthermore, the toolkit can be run periodically to detect fail-pass activities, thus growing the dataset continually.**

***Index Terms*—Bug Database, Reproducibility, Software Testing, Program Analysis, Experiment Infrastructure**

I. INTRODUCTION

Software defects have major impacts on the economy, on safety, and on the quality of life. Diagnosis and repair of software defects consumes a great deal of time and money. Defects can be treated more effectively, or avoided, by studying past defects and their repairs. Several software engineering subfields, e.g., program analysis, testing, and automatic program repair, are dedicated to developing tools, models, and methods for finding and repairing defects. These approaches, ideally, should be evaluated on realistic, up-to-date datasets of defects so that potential users have an idea of how well they work. Such datasets should contain *fail-pass pairs*, consisting of a failing version, which may include a test set that exposes the failure, and a passing version including changes that repair it. Given this, researchers can evaluate the effectiveness of tools that perform fault detection, localization (static or dynamic), or fault repair. Thus, research progress is intimately dependent on high-quality datasets of fail-pass pairs.

There are several desirable properties of these datasets of fail-pass pairs. First, **scale**: enough data to attain statistical significance on tool evaluations. Second, **diversity**: enough variability in the data to control for factors such as project scale, maturity, domain, language, defect severity, age, etc., while still retaining enough sample size for sufficient experimental power. Third, **realism**: defects reflecting actual fixes made by real-world programmers to repair real mistakes. Fourth, **currency**: a continuously updated defect dataset, keeping up with changes in languages, platforms, libraries, software function, etc., so that tools can be evaluated on bugs of current interest and relevance. Finally, and most crucially, defect data should be **durably reproducible**: defect data preserved in a way that supports durable build and behavior reproduction, robust to inevitable changes to libraries, languages, compilers, related dependencies, and even the operating system.[1]

Some hand-curated datasets (e.g., Siemens test suite [23], the SIR repository [21], Defects4J [24]) provide artifact collections to support controlled experimentation with program analysis and testing techniques. However, these collections are curated by hand, and are necessarily quite limited in *scale* and *diversity*; others incorporate small-sized student homeworks [25], which may not reflect development by professionals. Some of these repositories often rely on seeded faults; natural faults, from real programmers, would provide more *realism*. At time of creation, these are (or rather were) current. However, unless augmented through continuous and expensive manual labor, *currency* will erode. Finally, to the extent that they have dependencies on particular versions of libraries and operating systems, their future *reproducibility* is uncertain.

The datasets cited above have incubated an impressive array of innovations and are well-recognized for their contribution to research progress. However, we believe that datasets of greater scale, diversity, realism, currency, and durability will lead to even greater progress. The ability to control for covariates, without sacrificing experimental power, will help tool-builders and empirical researchers obtain results with greater discernment, external validity, and temporal stability. However, how can we build larger defect datasets without heavy manual labor? Finding specific defect occurrences, and creating recompilable and runnable versions of failing and passing software is difficult for all but the most trivial systems: besides the source code, one may also need to gather specific versions of libraries, dependencies, operating systems, compilers, and

[1]While it is impossible to guarantee this in perpetuity, we would like to have some designed-in resistance to change.

other tools. This process requires a great deal of human effort. Unless this human effort can somehow be automated away, we cannot build large-scale, diverse, realistic datasets of reproducible defects that continually maintain currency. But how can we automate this effort?

We believe that the DevOps- and OSS-led innovations in cloud-based continuous integration (CI) hold the key. CI services, like TRAVIS-CI [13], allow open-source projects to outsource integration testing. OSS projects, for various reasons, have need for continuous, automated integration testing. In addition, modern practices such as test-driven development have led to much greater abundance of automated tests. Every change to a project can be intensively and automatically tested off-site, on a cloud-based service; this can be done continually, across languages, dependencies, and runtime platforms. For example, typical GITHUB projects require that each pull request (PR) be integration tested, and failures fixed, before being vetted or merged by integrators [22, 32]. In active projects, the resulting back-and-forth between PR contributors and project maintainers naturally creates many fail-pass pair records in the pull request history and overall project history.

Two key technologies underlie this capability: *efficient, customizable, container-based virtualization* simplifies handling of complex dependencies, and *scripted CI servers* allows custom automation of build and test procedures. Project maintainers create scripts that define the test environment (platforms, dependencies, etc.) for their projects; using these scripts, the cloud-based CI services construct virtualized runtimes (typically DOCKER containers) to build and run the tests. The CI results are archived in ways amenable to mining and analysis. We exploit precisely these CI archives, and the CI technology, to create an automated, continuously growing, large-scale, diverse dataset of realistic and durably reproducible defects.

In this paper, we present BUGSWARM, a CI harvesting toolkit, together with a large, growing dataset of durably reproducible defects. The toolkit enables maintaining currency and augmenting diversity. BUGSWARM exploits archived CI log records to create detailed artifacts, comprising buggy code versions, failing regression tests, and bug fixes. When a successive pair of commits, the first, whose CI log indicates a failed run, and the second, an immediately subsequent passing run, is found, BUGSWARM uses the project's CI customization scripts to create an artifact: a fully containerized virtual environment, comprising both versions and scripts to gather all requisite tools, dependencies, platforms, OS, etc. BUGSWARM artifacts allow full build and test of pairs of failing/passing runs. Containerization allows these artifacts to be durably reproducible. The large scale and diversity of the projects using CI services allows BUGSWARM to also capture a large, growing, diverse, and current collection of artifacts.

Specifically, we make the following contributions:

- We present an approach that leverages CI to mine fail-pass pairs in open source projects and automatically attempts to reproduce these pairs in DOCKER containers (Section III).
- We show that fail-pass pairs are frequently found in open source projects and discuss the challenges in reproducing such pairs (Section IV).
- We provide the BUGSWARM dataset of 3,091 artifacts, for Java and Python, to our knowledge the *largest*, *continuously expanding*, *durably reproducible* dataset of fail-pass pairs, and describe the general characteristics of the BUGSWARM artifacts (Section IV).[2]

We provide background and further motivation for BUGSWARM in Section II. We describe limitations and future work in Section V. Finally, we discuss related work in Section VI and conclude in Section VII.

II. BACKGROUND AND MOTIVATION

Modern OSS development, with CI services, provides an enabling ecosystem of tools and data that support the creation of BUGSWARM. Here we describe the relevant components of this ecosystem and present a motivating example.

A. The Open-Source CI Ecosystem

GIT and GITHUB. GIT is central to modern software development. Each project has a *repository*. Changes are added via a *commit*, which has a unique identifier, derived with a SHA-1 hash. The project history is a sequence of commits. GIT supports branching. The main development line is usually maintained in a branch called `master`. GITHUB is a web-based service hosting GIT repositories. GITHUB offers *forking* capabilities, i.e., cloning a repository but maintaining the copy online. GITHUB supports the *pull request* (PR) development model: project maintainers decide on a case-by-case basis whether to accept a change. Specifically, a potential contributor forks the original project, makes changes, and then opens a pull request. The maintainers review the PR (and may ask for additional changes) before the request is merged or rejected.

TRAVIS-CI Continuous Integration. TRAVIS-CI is the most popular cloud-hosted CI service that integrates with GITHUB; it can automatically build and test commits or PRs. TRAVIS-CI is configured via settings in a `.travis.yml` file in the project repository, specifying all the environments in which the project should be tested. A TRAVIS-CI *build* can be initiated by a *push event* or a *pull request event*. A push event occurs when changes are pushed to a project's remote repository on a branch monitored by TRAVIS-CI. A pull request event occurs when a PR is opened and when additional changes are committed to the PR. TRAVIS-CI builds run a separate job for each configuration specified in the `.travis.yml` file. The build is marked as "passed" when all its jobs pass.

DOCKER. DOCKER is a lightweight virtual machine service that provides application isolation, immutability, and customization. An application can be packaged together with code, runtime, system tools, libraries, and OS into an immutable, stand-alone, custom-built, persistent DOCKER *image* (*container*), which can be run anytime, anywhere, on any platform that supports DOCKER. In late 2014, TRAVIS-CI began running builds and tests inside DOCKER containers, each customized for a specific run, as specified in the TRAVIS-

[2]The BUGSWARM dataset is available at http://www.bugswarm.org.

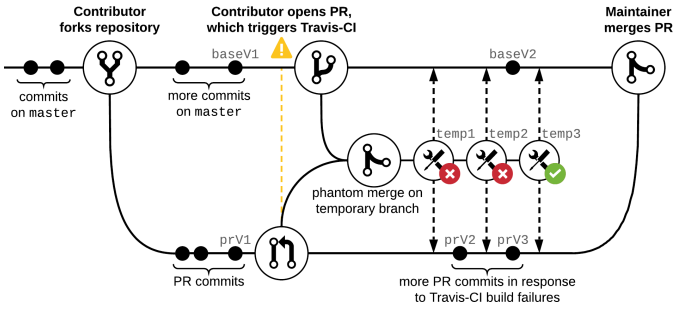


Fig. 1: Lifecycle of a TRAVIS-CI-built and tested PR

CI `.travis.yml` files. TRAVIS-CI maintains some of its *base* images containing a minimal build environment. BUGSWARM harvests these containers to create the dataset.

B. Leveraging TRAVIS-CI to Mine and Reproduce Bugs

We exploit TRAVIS-CI to create BUGSWARM. Figure 1 depicts the lifecycle of a TRAVIS-CI-built and tested PR. A contributor forks the repository and adds three commits, up to `prV1`; she then opens a PR, asking that her changes be merged into the original repository. The creation of the PR triggers TRAVIS-CI, which checks whether there are merge conflicts between the PR branch and `master` when the PR was opened (`prV1` and `baseV1`). If not, TRAVIS-CI creates a *temporary* branch from the base branch, into which the PR branch is merged to yield `temp1`. This merge is also referred to as a "phantom" merge because it disappears from the GIT history after some time.[3] TRAVIS-CI then generates build scripts from the `.travis.yml` file and initiates a build, i.e., runs the scripts to compile, build, and test the project.

In our example, test failures cause the first build to fail; TRAVIS-CI notifies the contributor and project maintainers, as represented by the dashed arrows in Figure 1. The contributor does her fix and updates the PR with a new commit, which triggers a new build. Again, TRAVIS-CI creates the merge between the PR branch (now at `prV2`) and the base branch (still at `baseV1`) to yield `temp2`. The build fails again; apparently the fix was no good. Consequently, the contributor updates the PR by adding a new commit, `prV3`. A TRAVIS-CI build is triggered in which the merge (`temp3`) between the PR branch (at `prV3`) and the base branch (now at `baseV2`) is tested.[4] This time, the build passes, and the PR is accepted and merged into the base branch.

Each commit is recorded in version control, archiving source code at build-time plus the full build configuration (`.travis.yml` file). TRAVIS-CI records how each build fared (pass or fail) and archives a build log containing output of the build and test process, including the names of any failing tests.

Our core idea is that TRAVIS-CI-built and tested pull requests (and regular commits) from GITHUB, available in large volumes for a variety of languages and platforms, can be used to construct *fail-pass pairs*. In our example, the version of the code represented by the merge `temp2` is "defective," as documented by test failures in the corresponding TRAVIS-CI build log. The subsequently "fixed" version (no test failures in the build log) is represented by `temp3`. Therefore, we can extract (1) a failing program version; (2) a subsequent, fixed program version; (3) the fix, i.e., the difference between the two versions; (4) the names of failing tests from the failed build log; (5) a full description of the build configuration.

Since each TRAVIS-CI job occurs within a DOCKER container, we can re-capture that specific container image, thus rendering the event durably reproducible. Furthermore, if one could build an *automated harvesting system* that could continually mine TRAVIS-CI builds and create DOCKER images that could persist these failures and fixes, this promises a way to create a dataset to provide all of our desired data: GITHUB-level *scale*; GITHUB-level *diversity*; *realism* of popular OSS projects; *currency* via the ability to automatically and periodically augment our dataset with recent events, and finally *durable reproducibility* via DOCKER images.

III. BUGSWARM INFRASTRUCTURE

A. Some Terminology

A project's *build history* refers to all TRAVIS-CI builds previously triggered. A *build* may include many *jobs*; for example, a build for a Python project might include separate jobs to test with Python versions 2.6, 2.7, 3.0, etc.

A *commit pair* is a 2-tuple of GIT commit SHAs that each triggered a TRAVIS-CI build in the *same* build history. The canonical commit pair consists of a commit whose build fails the tests followed by a fix commit whose build passes the tests. The terms *build pair* and *job pair* refer to a 2-tuple of TRAVIS-CI builds or jobs, respectively, from a project's build history. For a given build, the *trigger commit* is the commit that, when pushed to the remote repository, caused TRAVIS-CI to start a build.

BUGSWARM has four components: PAIRMINER, PAIRFILTER, REPRODUCER, and ANALYZER. These components form the pipeline that curates BUGSWARM artifacts and are designed to be relatively independent and general. This section describes the responsibilities and implementation of each component, and a set of supporting tools that facilitate usage of the dataset.

B. Design Challenges

The tooling infrastructure is designed to handle certain specific challenges, listed below, that arise when one seeks to continuously and automatically mine TRAVIS-CI. In each case, we list the tools that actually address the challenges.

Pair coherence. Consecutive commits in a GIT history may not correspond to consecutive TRAVIS-CI builds. A build history, which TRAVIS-CI retains as a linear series of builds, must be traversed and transformed into a directed graph so that pairs of consecutive builds map to pairs of consecutive commits. GIT's non-linear nature makes this non-trivial. (PAIRMINER)

[3] "Phantom" merges present special challenges, which are discussed later.

[4] TRAVIS-CI creates each phantom merge on a separate temporary branch, but Figure 1 shows the phantom merges on a single branch for simplicity.

Algorithm 1: PAIRMINER Algorithm

Input: Project slug P
Output: Set J of fail-pass job pairs (j_f, j_p)

```
J = ∅  B = the list of TRAVIS-CI builds for P;
G = {g | g ⊆ B and ∀b ∈ g belong to the same branch/PR};
foreach g in G do
    Order the builds in g chronologically;
    foreach b_i ∈ g do
        if b_i is failed and b_{i+1} is passed then
            AssignCommits(b_i);
            AssignCommits(b_{i+1});
            J = J ∪ {(j_f, j_p) | j_f ∈ b_i and j_p ∈ b_{i+1} and j_f
              has the same configuration as j_p};
return J
```

Algorithm 2: AssignCommits Algorithm

Input: TRAVIS-CI build B

```
Mark B as "unavailable" by default;
Clone the GIT repository for B;
if B is triggered by a push event then
    Assign trigger commit t from TRAVIS-CI build metadata;
    if t in GIT history or t in GITHUB archive then
        mark B as "available";
else if B is triggered by a pull request event then
    Assign trigger commit t, base commit b, and merge
      commit m for B from TRAVIS-CI build metadata;
    if t and b in GIT history or m in GITHUB archive then
        mark B as "available";
```

Commit recovery. To reproduce a build, one needs to find the trigger commit. There are several (sub-)challenges here. First, temporary merge commits like `temp1,2,3` in Figure 1 are the ones we need to extract, but these are not retained by TRAVIS-CI. Second, GIT's powerful history-rewriting capabilities allow commits to be erased from history; developers can and do collapse commits like `prV1,2,3` into a single commit, thus frustrating the ability to recover the consequent phantom merge commits. (PAIRMINER, PAIRFILTER)

Image recovery. In principle, TRAVIS-CI creates and retains DOCKER images that allow re-creation of build and test events. In practice, these images are not always archived as expected and so must be reconstructed. (PAIRFILTER, REPRODUCER)

Runtime recovery. Building a specific project version often requires satisfying a large number of software dependencies on tools, libraries, and frameworks; all or some of these may have to be "time-traveled" to an earlier version. (REPRODUCER)

Test flakiness. Even though TRAVIS-CI test behavior is theoretically recoverable via DOCKER images, tests may behave non-deterministically because of concurrency or environmental (e.g., external web service) changes. Such flaky tests lead to flaky builds, which both must be identified for appropriate use in experiments. (REPRODUCER)

Log analysis. Once a pair is recoverable, BUGSWARM tries to determine the exact nature of the failure from the logs, which are not well structured and have different formats for each language, build system, and test toolset combination. Thus the logs must be carefully analyzed to recover the nature of the failure and related metadata (e.g., raised exceptions, failed test names, etc.), so that the pair can be documented. (ANALYZER)

C. Mining Fail-Pass Pairs

PAIRMINER extracts from a project's GIT and build histories a set of fail-pass job pairs (Algorithm 1). PAIRMINER takes as input a GITHUB slug and produces a set of fail-pass job pairs annotated with trigger commit information for each job's parent build. The PAIRMINER algorithm involves (1) delinearizing the project's build history, (2) extracting fail-pass build pairs, (3) assigning commits to each pair, and (4) extracting fail-pass job pairs from each fail-pass build pair.

Analyzing build history. PAIRMINER first downloads the project's entire build history with the TRAVIS-CI API. For each build therein, PAIRMINER notes the branch and (if applicable) the pull request containing the trigger commit. PAIRMINER first resolves the build history into lists of builds that were triggered by commits on the same branch or pull request. PAIRMINER recovers the results of the build and its jobs (passed, failed, errored, or canceled), the `.travis.yml` configuration of each job, and the unique identifiers of the build and its jobs using the TRAVIS-CI API.

Identifying fail-pass build pairs. Using the build and job identifiers, PAIRMINER finds consecutive pairs where the first build failed and the second passed. Builds are considered from all branches, including the main line and any perennials, and both merged and unmerged pull requests. Next, the triggering commits are found, and recovered from GIT history.

Finding trigger commits. If the trigger commit was a push event, then PAIRMINER can find its SHA via the TRAVIS-CI API. For pull request triggers, we need to get the pull request and base branch head SHAs, and re-create the phantom merge. Unfortunately, neither the trigger commit nor the base commit are stored by TRAVIS-CI; recreating them is quite a challenge. Fortunately, the commit message of the phantom commit, which is stored by TRAVIS-CI, contains this information; we follow Beller et al. [17] to extract this information. This approach is incomplete but is the best available.

TRAVIS-CI creates temporary merges for pull-request builds. While temporary merges may no longer be directly accessible, the information for such builds (the head SHAs and base SHAs of the merges) are accessible through the GITHUB API. We resort to GITHUB archives to retrieve the code for the commits that are no longer in GIT history.

Even if the trigger commit is recovered from the phantom merge commit, one problem remains: developers might squash together all commits in a pull request, thus erasing the constituent commits of the phantom merge right out of the GIT history. In addition, trigger commits for push event builds can sometimes also be removed from the GIT history by the project personnel. As a result, recreating this merge is not always possible; we later show the proportion for which we

were able to reset the repository to the commits in the fail-pass pairs. The two steps of phantom recovery—first finding the trigger commits and then ensuring that the versions are available in GITHUB —are described in Algorithm 2.

Extracting fail-pass job pairs. PAIRMINER now has a list of fail-pass build pairs for the project. As described in Section II, each build can have many jobs, one for each supported environment. A build fails if any one of its jobs fails and passes if all of its jobs pass. Given a failing build, PAIRMINER finds pairs of jobs, *executed in the same environment*, where the first failed and the second passed. Such a pair only occurs when a defective version was fixed via source code patches and not by changes in the execution environment (see Algorithm 1).

D. Finding Essential Information for Reproduction

Pairs identified by PAIRMINER must be assembled into reproducible containers. To stand a chance of reproducing a job, one must have access to, at a minimum, these essentials: (1) the state of the project at the time the job was executed and (2) the environment in which the job was executed. For each job in the pipeline, PAIRFILTER checks that these essentials can be obtained. If the project state was deemed recoverable by PAIRMINER, PAIRFILTER retrieves the original TRAVIS-CI log of the job and extracts information about the execution environment. Using timestamps and instance names in the log, PAIRFILTER determines if the job executed in a DOCKER container and, if so, whether the corresponding image is still accessible. If the log is unavailable, the job was run before TRAVIS-CI started using DOCKER, or the particular image is no longer publicly accessible, then the job is removed from the pipeline.

E. Reproducing Fail-Pass Pairs

REPRODUCER checks if each job is durably reproducible. This takes several steps, described below.

Generating the job script. `travis-build`,[5] a component of TRAVIS-CI, produces a shell script from a `.travis.yml` file for running a TRAVIS-CI job. REPRODUCER then alters the script to reference a specific past version of the project, rather than the latest.

Matching the environment. To match the original job's runtime environment, REPRODUCER chooses from the set of TRAVIS-CI's publicly available DOCKER images, from Quay and DockerHub, based on (1) the language of the project, as indicated by its `.travis.yml` configuration, and (2) a timestamp and instance name in the original job log that indicate when that image was built with DOCKER's tools.

Reverting the project. For project history reversion, REPRODUCER clones the project and resets its state using the trigger commit mined by PAIRMINER. If the trigger was on a pull request, REPRODUCER re-creates the phantom merge commit using the trigger and base commits mined by PAIRMINER. If any necessary commits were not found during the mining process, REPRODUCER downloads the desired state of the project directly from a zip archive maintained by GITHUB.[6] Finally, REPRODUCER plants the state of the project inside the execution environment to reproduce the job.

[5] https://github.com/travis-ci/travis-build

TABLE I: BUGSWARM's main metadata attributes

Attribute Type	Description
Project	GITHUB slug, primary language, build system, and test framework
Reproducibility	Total number of attempts, and number of successful attempts to reproduce pair
Pull Request	Pull request #, merge timestamp, and branch
TRAVIS-CI Job	TRAVIS-CI build ID, TRAVIS-CI job ID, number of executed and failed tests, names of the failed tests, trigger commit, and branch name
Image Tag	Unique *image tag* (simultaneously serves as a reference to a particular DOCKER image)

Reproducing the job. REPRODUCER creates a new DOCKER image, as described in Section III-E, runs the generated job script, and saves the resulting output stream in a log file. REPRODUCER can run multiple jobs in parallel. REPRODUCER collects the output logs from all the jobs it attempts to reproduce and sends them to ANALYZER for parsing.

F. Analyzing Results

ANALYZER parses a TRAVIS-CI build log to learn the status of the build (passed, failed, etc.) and the result of running the regression test suite. If there are failing tests, then ANALYZER also retrieves their names. A challenge here: the format of build logs varies substantially with the specific build system and the test framework; so parsers must be specialized to each build and test framework. For Java, we support the most popular build systems—Maven [8], Gradle [6], and Ant [1]—and test frameworks—JUnit [7] and testng [12]. For Python, we support the most popular test frameworks—unittest [15], unittest2 [16], nose [9], and pytest [10].

ANALYZER has a top-level analyzer that retrieves all language-agnostic items, such as the operating system used for a build, and then delegates further log parsing to language-specific and build system-specific analyzers that extract information related to running the regression test suite. The extracted attributes—number of tests passed, failed, and skipped; names of the failed tests (if any); build system, and test framework—are used to compare the original TRAVIS-CI log and the reproduced log. If the attributes match, then we say the run is reproducible. Writing a new language-specific analyzer is relatively easy, mostly consisting of regular expressions that capture the output format of various test frameworks.

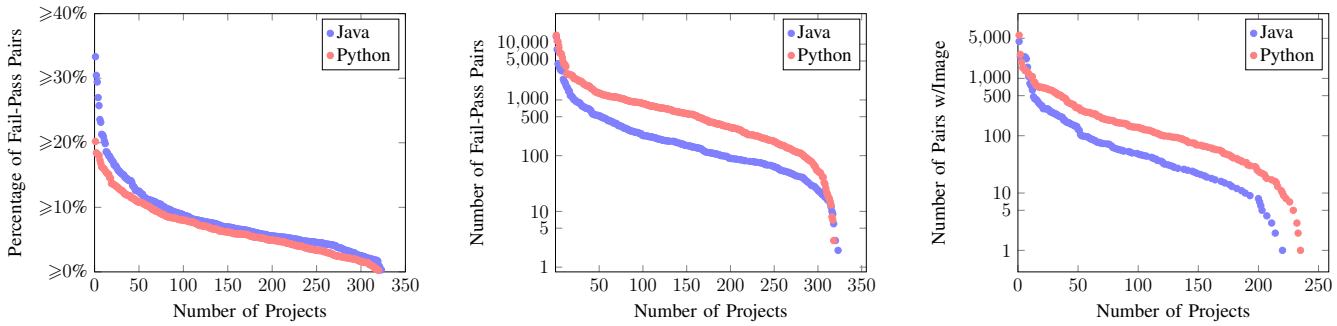
G. Tools for BUGSWARM Users

BUGSWARM includes tools to support tasks such as artifact selection, artifact retrieval, and artifact execution.

[6] GITHUB allows one to download a zip archive of the entire project's file structure at a specific commit. Since this approach produces a stand-alone checkout of a project's history (without any of the GIT data stores), REPRODUCER uses this archive only if a proper clone and reset is not possible.

TABLE II: Mined Fail-Pass Pairs

	Push Events					Pull Request Events				
Language	Failed Jobs	All Pairs	Available	DOCKER	w/Image	Failed Jobs	All Pairs	Available	DOCKER	w/Image
Java	320,918	80,804	71,036	50,885	29,817	250,349	63,167	24,877	20,407	9,509
Python	778,738	115,084	103,175	65,924	37,199	1,190,186	188,735	62,545	45,878	24,740
Grand Total	1,099,656	195,888	174,211	116,809	**67,016**	1,440,535	251,902	87,422	66,285	**34,249**



(a) Percent of Fail-Pass Pairs per Language (b) Cumulative Number of Fail-Pass Pairs (c) Cumulative Number of Pairs w/ Image

Fig. 2: Frequency of Fail-Pass Pairs

Artifact selection & retrieval. A given experiment may require artifacts meeting specific criteria. For this reason, each artifact includes metadata as described in Table I. The BUGSWARM website provides an at-a-glance view of the metadata for all artifacts. Simple filtering can be done directly via the web interface. For more advanced filtering, we provide a REST API; a Python API is also available.

To facilitate retrieval of artifact DOCKER images, we provide a BUGSWARM command line interface that masks the complexities of the DOCKER ecosystem to use our artifacts. Given any BUGSWARM artifact identifier, the CLI can download the artifact image, start an interactive shell inside the container, and clean up the container after use.[7]

Artifact execution. A typical workflow for experiments with BUGSWARM involves copying tools and scripts into a container, running jobs, and then copying results. We provide a framework to support this common artifact processing workflow. The framework can be extended to fit users' specific needs. See the BUGSWARM website for example applications.

IV. EXPERIMENTAL EVALUATION

Our evaluation is designed to explore the feasibility of automatically creating a large-scale dataset of reproducible bugs and their corresponding fixes. In particular, we answer the following research questions:

RQ1: How often are fail-pass pairs found in OSS projects?
RQ2: What are the challenges in automatically reproducing fail-pass pairs?
RQ3: What are the characteristics of reproducible pairs?

The BUGSWARM infrastructure is implemented in Python. REPRODUCER uses a modified version of the `travis-build` component from TRAVIS-CI to translate `.travis.yml` files into shell scripts. The initial Java-specific ANALYZER was ported to Python from TRAVISTORRENT's [17] implementation in Ruby. ANALYZER has been extended to support JUnit for Java and now also supports Python.

BUGSWARM requires that a project is hosted on GITHUB and uses TRAVIS-CI. We randomly selected 335 projects among the 500 GITHUB projects with the most TRAVIS-CI builds, for each of Java and Python.

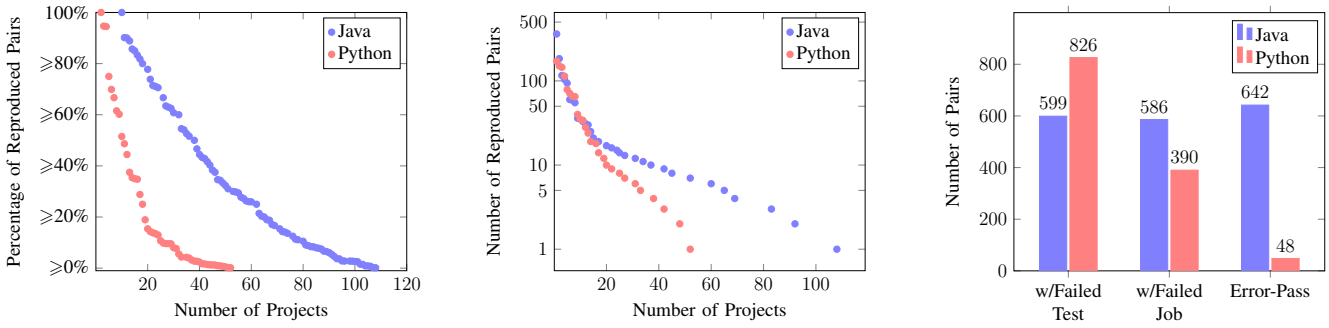
A. Mining Fail-Pass Pairs

We inspected a total of 10,179,558 jobs across 670 projects, from which 2,540,191 are failed jobs. We mined a total of 447,790 fail-pass pairs. As described in Section III-C, pairs can originate from push events or pull request events. Table II shows the breakdown: push events contribute to 44% of the fail-pass pairs (195,888) and pull requests contribute to 56% (251,902). Note that fail-pass pairs represent an under-approximation of the number of bug-fix commits; BUGSWARM pairs do not capture commits that fix a bug whose build is not broken. We calculate the percentage of fail-pass pairs with respect to the total number of successful jobs (potential fixes to a bug) per project. Figure 2a plots a cumulative graph with the results. In general, we find that Java projects have a slightly higher percentage of fail-pass pairs (at most 33%) than Python projects (at most 20%). For example, there are 80 Java projects and 61 Python projects for which at least 10% of the passing jobs fix a build. Figure 2b plots the cumulative *number* of fail-pass pairs per project. The Java and Python projects with the most pairs have 13,699 and 14,510 pairs, respectively.

[7] https://github.com/BugSwarm/client

TABLE III: Reproduced Pairs

Language	Pairs to Reproduce	Fully Reproducible + Flaky				Unreproducible	Pending
		w/Failed Test	w/Failed Job	Error-Pass	Total Pairs		
Java	39,326	584 + 15	564 + 22	626 + 16	1,827	17,369	20,130
Python	61,939	785 + 41	387 + 3	48 + 0	1,264	35,126	25,549
Grand Total	101,265	1,425	976	690	**3,091**	52,495	45,679



(a) Cumulative Percentage of Reprod. Pairs (b) Cumulative Number of Reprod. Pairs (c) Breakdown of Reproduced Pairs

Fig. 3: Reproduced Pairs

We run PAIRFILTER to discard fail-pass pairs that are unlikely to be reproducible. Table II shows the number of fail-pass pairs after each filter is applied. Specifically, columns "Available" show the pairs we can reset to or which are archived, columns "DOCKER" show the number of remaining pairs that use a DOCKER image, and columns "w/Image" show the number of remaining pairs for which we can locate TRAVIS-CI base images. Figure 2c plots the cumulative *number* of w/Image pairs, which are passed to REPRODUCER. A total of 220 Java projects and 233 Python projects have w/Image pairs.

RQ1: At most 33% and 22% of all pairs of Java and Python projects, respectively, follow the fail-pass pattern (Figure 2). Among 670 projects, we find a total of 447,490 fail-pass pairs, from which 101,265 pairs may be reproducible.

B. Reproducing Fail-Pass Pairs

We successfully reproduced 3,091 out of 55,586 attempted pairs (45,679 pairs are pending reproduction due to time constraints). Recall from Section III-C that PAIRMINER mines *job pairs*. The corresponding number of reproducible unique *build pairs* is 1,837 (1,061 for Java and 776 for Python). The rest of the paper describes the results in terms of number of job pairs. The 3,091 artifacts belong to 108 Java projects and 52 Python projects. Table IV lists the 5 projects with the most artifacts for each language. We repeated the reproduction process 5 times for each pair to determine its stability. If the pair is reproducible all 5 times, then it is marked as "reproducible." If the pair is reproduced only sometimes, then it is marked as "flaky." Otherwise, the pair is said to be "unreproducible." Numbers for each of these categories can be found in Table III.

Figure 3a shows the cumulative percentage of reproduced pairs across projects. We achieve a 100% pair reproduction rate for 10 Java projects and 2 Python projects, at least 50% for 38 Java projects and 50 Python projects, and at least 1 pair is reproducible in 108 Java projects, and 52 Python projects. Figure 3b shows the cumulative *number* of reproduced pairs. The Java and Python projects with the most reproducible pairs have 361 and 171, respectively.

We further classify "reproducible" and "flaky" pairs into three groups: (1) pairs that have failed tests, (2) pairs that do not have failed tests despite a failed build, and (3) pairs whose build finishes with an error. (1) and (2) are labeled *failed* and (3) *errored*. This naming convention is from TRAVIS-CI [14] and is defined by the part of the job lifecycle that encounters a non-zero exit code. Typically, *errored* builds have dependency-related issues. Figure 3c shows the breakdown for both Java and Python. We find that 46.1%, 31.6%, and 22.3% of reproducible pairs correspond to each of the above categories, respectively.

Surprisingly, only 97 pairs were "flaky." We suspect a number of unreproducible pairs are indeed flaky but running them 5 times was not sufficient to identify them. We plan to investigate how to grow the number of flaky pairs in BUGSWARM. An initial direction could involve selecting pairs based on keywords in their commit messages (e.g., [27]).

Among all the pairs that we attempted to reproduce, most were not reproducible. In other words, the log of the original job and the log produced by REPRODUCER were different. To gather information about the causes of unreproducibility, we randomly sampled 100 unreproducible job pairs and manually inspected their 200 logs (two logs per job pair). For this task, we also examined the corresponding 200 original logs

TABLE IV: Top Projects with Artifacts

Java	# Pairs	Python	# Pairs
raphw/byte-buddy	361	terasolunaorg/guideline	171
checkstyle/checkstyle	184	scikit-learn/scikit-learn	151
square/okhttp	104	numpy/numpy	145
HubSpot/Baragon	94	python/mypy	114
tananaev/traccar	59	marshallward/f90nml	65

TABLE V: Sources of Unreproducibility

Reason	# Pairs
Failed to install dependency	59
URL no longer valid or network issue	57
TRAVIS-CI command issue	38
Project-specific issue	22
REPRODUCER did not finish	14
Permission issue	4
Total	194

produced by TRAVIS-CI to compare the differences between logs and categorize the various sources of unreproducibility.

As shown in Table V, we identified 6 sources of unreproducibility. From the 200 jobs, around 30% are unreproducible due to missing or incompatible dependencies. Another 30% referenced stale URLs or experienced network issues. Exceptions from invoking `travis-build` when creating build scripts are responsible for another 20%. The rest of the jobs are unreproducible due to project-specific issues, failure to terminate within the time budget, or permission errors. Interestingly, 6 jobs are actually reproducible, but since the corresponding failed or passed job is not reproducible, the entire pair is marked as unreproducible. We have not included unreproducible pairs in this iteration of BUGSWARM, but we think these could also be potentially useful to researchers interested in automatically fixing broken builds.

RQ2: Reproducing fail-pass pairs is indeed challenging with a 5.56% success rate. Based on the manual inspection of 100 unreproducible artifacts, we identified 6 main reasons for unreproducibility listed in Table V.

C. General Characteristics of BUGSWARM Artifacts

We have aggregated statistics on various artifact characteristics. Figure 4a shows the number of artifacts with a given number of changes (additions or deletions). Inserting or removing a line counts as one change. Modifying an existing line counts as *two* changes (an addition and a deletion). Commits with zero changes are possible but rare and are not included in Figure 4a. We report the number of changes of the fixed version with respect to the failing version of the code, e.g., 31% (844) of the artifacts have at most 5 changes and 54% (1,458) have at most 20. Figure 4b shows the number of projects with a given number of files changed, e.g., 46% (1,335) of the artifacts have at most 5 changed files. Figure 4c shows the artifacts with a given number of failed tests.

We find that our artifacts are diverse in several aspects:

TABLE VI: Diversity of Artifacts

Type	# Artifacts	Type	# Artifacts
Language		Longetivity	
Java	1,827	2015; 2016	790; 989
Python	1,264	2017; 2018	807; 515
Build System		Test Framework	
Maven	1,675	JUnit	768
Gradle	86	unittest	665
Ant	66	Others	1,415

language, build system, test framework, and longevity. Table VI shows the number of reproducible and flaky artifacts for each of these categories. The current dataset has over a thousand artifacts for Java and Python with a wide range of build systems and testing frameworks being used. From these, the most common build system is Maven with 1,675 artifacts, and the most common testing framework is JUnit with 768. We plan to add support for other languages such as JavaScript and C++ in the near future, which will increase the number of build systems and testing frameworks being used.

Our artifacts represent a variety of software bugs given the diverse set of projects mined. To better understand the types of bugs in BUGSWARM, we conduct a manual classification of 320 randomly sampled Maven-based Java artifacts, first described in [30]. The top 10 classification categories are shown in Figure 5a. The classification is not one-to-one; an artifact may fall under multiple categories depending on the bug. To correctly classify an artifact, we examine the source code, diff, commit message, and TRAVIS-CI log. We find that the largest category is logic errors. Examples of logic errors include off-by-one errors and incorrect logical operations.

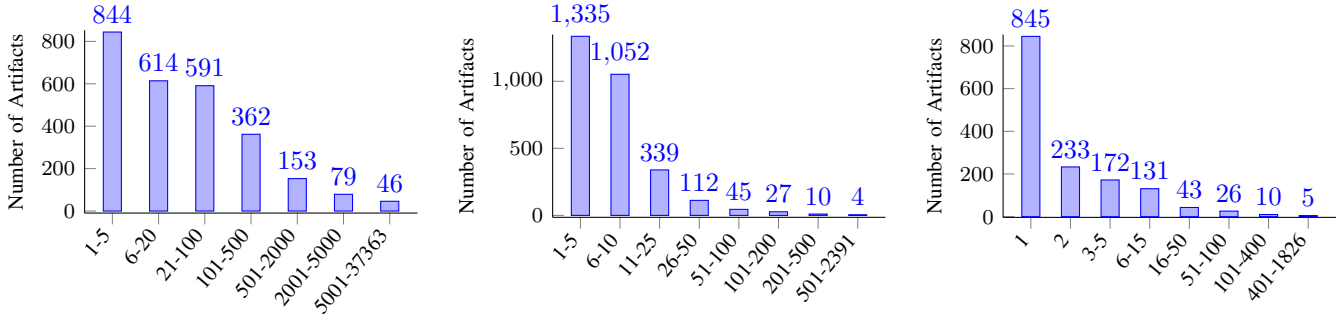
We also conduct an automatic higher-level classification of artifacts based on the encountered exceptions or runtime errors. We analyze the build logs and search for the names of Java exceptions and Python runtime errors. Figures 5b and 5c show the 10 exceptions/errors for which BUGSWARM has the most artifacts. For example, 252 Java artifacts fail with a NullPointerException. An example is shown in Figure 6.

Using the BUGSWARM framework presented in Section III-G, we successfully ran the code coverage tool Cobertura [3] and two static analyzers—Google's ErrorProne [5] and SpotBugs [11]—on the 320 randomly selected artifacts used in the manual classification [30], with minimal effort.

RQ3: We investigated various characteristics of artifacts, such as the distribution in the size of the diff, location of the diff, and number of failing tests (Figure 4). We also examined the reason for failure (Figure 5). For example, 844 artifacts have between 1 and 5 changes, 1,335 artifacts modify a single file, 845 artifacts have 1 failing test, and the top reason for a build failure is an AssertionError.

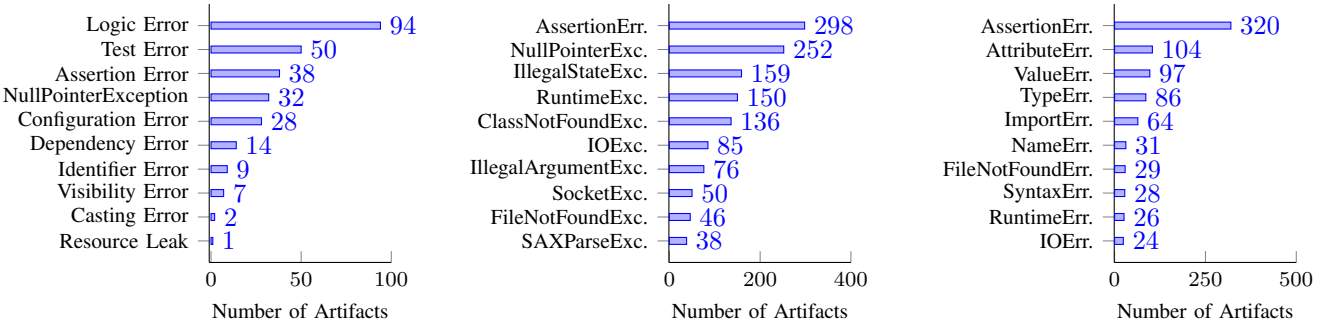
D. Performance

PAIRMINER and REPRODUCER can be run in the cloud in parallel. The BUGSWARM infrastructure provides support to run these as batch tasks on Microsoft Azure [2]. Running time of PAIRMINER depends on the number of failed jobs



(a) Number of changes (b) Number of files changed (c) Number of failing tests

Fig. 4: Artifact Characteristics



(a) Manual Classification of Java Bugs (b) Most Frequent Java Exceptions (c) Most Frequent Python Errors

Fig. 5: Artifact Classification

```
  protected void loadCommandVerificationSheet(
      SpaceSystem spaceSystem, String sheetName) {
    Sheet sheet = switchToSheet(sheetName, false);
+   if (sheet == null) return;
    int i = 1;
    while (i < sheet.getRows()) {
      // search for a new command definition
      ...
    }
  }
```

Fig. 6: Example of NullPointerException bug and its fix.

to examine, taking between a few minutes to a few hours. Reproduction time varies per project as it depends on the project's build time and the number of tests run. Mining and reproducing the pairs reported in this paper required about 60,000 hours of compute time in Azure. We will continue our effort to mine and reproduce pairs in additional projects.

V. Limitations and Future Work

PairMiner searches for two consecutive failed and passed builds first then looks for failed and passed job pairs within these two builds. However, failed and passed job pairs can occur between two consecutive failed builds because a build marked as failed requires only one unsuccessful job. In addition, the fail-pass pattern does not guarantee that the difference between the two commits is actually a fix for the failure; the supposed fix could simply delete or revert the buggy code or disable any failing tests. Using only the pattern, PairMiner would also fail to identify a fix for a failure if the fix is committed along with the test cases that expose the fail point. Finally, the fix may not be minimal. We plan to address some of these challenges in the future. In particular, we would like to explore other mining approaches that involve new patterns as well as bug reports. Note that Reproducer is already capable of reproducing any pair of commits that triggered Travis-CI builds, regardless of how these commits are gathered.

Reproducible artifacts may still break later on due to stale URLs, among other reasons. To keep BugSwarm up to date, we periodically test artifacts. We are currently exploring ways to make the artifacts more robust. In the future, we would like to crowdsource the maintainability of BugSwarm.

Thus far, our mining has been "blind." However, it is possible to extend our mining tools to find pairs with specific characteristics (e.g., pairs that have at most 5 changes and a single failed test caused by a NullPointerException). Such guided mining will allow BugSwarm to grow in directions of interest to the research community. Finally, we plan to extend BugSwarm to continuously monitor Travis-CI events for real-time mining and reproducing of new artifacts.

VI. Related Work

Some other defect repositories aim to provide experimental benchmarks for defect location and repair. On the whole, these

repositories do not exploit CI and virtualization mechanisms; they generally pre-date the widespread adoption of these techniques. They do not achieve the same *scale*, *diversity*, and *currency* and are not as *durably reproducible*.

The Siemens test suite [23] (7 small C programs and about 130 manually seeded bugs) is among the earliest. BugBench [26] is one of the earliest datasets of real-world bugs. BugBench is limited in scale and diversity, consisting of 7 memory and concurrency bugs found across 10 C/C++ OS projects. Each buggy program version includes failing tests. BegBunch [18] contains two suites to measure the accuracy and scalability of bug detection tools for C. iBugs [19] is a dataset drawn from the 5-year history of the AspectJ compiler with 369 faulty versions of the project. iBugs provides metadata such as number of methods and classes involved in the bug fix. Unlike BUGSWARM, the above datasets were manually constructed. Metadata such as that included in iBugs could be built from BUGSWARM artifacts with additional effort.

The Software-artifact Infrastructure Repository (SIR) [21] comprises source code, tests, and defects from OS projects along with needed infrastructure (e.g., automated build and test scripts). Currently, SIR consists of 85 projects in four languages, of which 64 (15 C, 1 C#, 1 C++, and 47 Java) include fault data: real ones; seeded ones; and a combination of real, seeded, and mutated. A project may contain multiple versions, and each version may contain multiple faults, with a total of 680 bugs. SIR provides a useful amount of scale and diversity while archiving sufficient tooling for durable reproducibility. However, since it pre-dates CI and DOCKER, each defect datum therein is manually assembled. Thus, SIR is difficult to scale up further and requires substantial effort to keep current. BUGSWARM already has 3,091 reproducible defects; the automated mining of CI and DOCKER image artifacts lowers the cost of keeping the dataset growing.

MANYBUGS [25] is a benchmark for program repair with 185 defects and fixes from 9 large C projects. Each defect and fix includes tests and is manually categorized. To facilitate the reproduction of these defects, MANYBUGS provides virtual machine images (recently extended to use DOCKER [29]). Unlike BUGSWARM, mining and reproducing bugs requires significant manual effort, and thus MANYBUGS is not as easy to extend. On the other hand, MANYBUGS provides a detailed bug categorization that can be useful for experiments, and its artifacts are collected from C programs, a programming language that BUGSWARM does not currently support.

Defects4J [24][4] is a dataset of 395 real, reproducible bugs from 6 large Java projects. Defects4J provides manually constructed scripts for each project's build and test; the entire setup relies on a functioning JVM. Defects4J provides an interface for common tasks and provides support for a number of tools. The Bugs.jar [28] dataset contains 1,158 real, reproducible Java bugs collected from 8 Apache projects by identifying commit messages that reference bug reports. Bugs.jar artifacts are stored on GIT branches. By contrast, BUGSWARM relies on virtualized, DOCKER-packaged build and test environments, automatically harvested from the cross-platform TRAVIS-CI archives; thus it is neither limited to Java nor does it require manual assembly of build and test tools. In addition to the test fail-pass pairs, we include build failures and even flaky tests. The above allows BUGSWARM to achieve greater scale, diversity, and currency.

Urli et al. [31] describe an approach to mining builds that fail tests from TRAVIS-CI. This work can only handle Maven-based Java builds; these are reproduced directly, without DOCKER. Their dataset includes 3,552 Maven Java builds for the purpose of automatic repair. Delfim et al. [20] develop BEARS, which mines Maven-based Java GITHUB projects that use TRAVIS-CI. BEARS attempts to reproduce every mined build in the same environment, which does not account for the developer-tailored `.travis.yml` file, whereas BUGSWARM leverages DOCKER images to match each job's original runtime environment. Compared to BUGSWARM, BEARS has a similar reproduction success rate of 7% (856 builds). BEARS pushes artifacts to GIT branches, instead of providing them as DOCKER images, and relies on Maven for building and testing, so new infrastructure must be implemented to include artifacts from other build systems.

Our DOCKER-based approach allows other languages and build systems, and reflects our designed-in pursuit of greater diversity and reproducibility. Note that the BUGSWARM toolset supports the creation of fully reproducible packages for any pair of commits for which the TRAVIS-CI builds are archived. There are over 900K projects in GITHUB that use TRAVIS-CI [13], so our toolkit enables the creation of datasets and ensuing experiments at a scale substantially larger than previous datasets allow.

VII. CONCLUSIONS

This paper described BUGSWARM, an approach that leverages CI to mine and reproduce fail-pass pairs of realistic failures and fixes in Java and Python OSS. We have already gathered 3,091 such pairs. We described several exciting future directions to further grow and improve the dataset. We hope BUGSWARM will minimize effort duplication in reproducing bugs from OSS and open new research opportunities to evaluate software tools and conduct large-scale software studies.

ACKNOWLEDGMENTS

We thank Christian Bird, James A. Jones, Claire Le Goues, Nachi Nagappan, Denys Poshyvanyk, Westley Weimer, and Tao Xie for early feedback on this work. We also thank Saquiba Tariq, Pallavi Kudigrama, and Bohan Xiao for their contributions to improve BUGSWARM and Aditya Thakur for feedback on drafts of this paper. This work was supported by NSF grant CNS-1629976 and a Microsoft Azure Award.

REFERENCES

[1] Apache Ant. http://ant.apache.org, Accessed 2019.
[2] Microsoft Azure. http://azure.microsoft.com, Accessed 2019.
[3] Cobertura. https://github.com/cobertura/cobertura/wiki, Accessed 2019.
[4] Defects4J. https://github.com/rjust/defects4j, Accessed 2019.
[5] Error Prone. https://github.com/google/error-prone, Accessed 2019.
[6] Gradle Build Tool. https://gradle.org, Accessed 2019.

[7] JUnit Test Framework. https://junit.org/junit5, Accessed 2019.

[8] Apache Maven Project. https://maven.apache.org, Accessed 2019.

[9] Test Framework nose. http://nose.readthedocs.io/en/latest, Accessed 2019.

[10] Test Framework pytest. https://docs.pytest.org/en/latest, Accessed 2019.

[11] SpotBugs Bug Descriptions. https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html, Accessed 2019.

[12] Test Framework testng. http://testng.org/doc, Accessed 2019.

[13] Travis CI. https://travis-ci.org, Accessed 2019.

[14] Job Lifecycle. https://docs.travis-ci.com/user/job-lifecycle/#breaking-the-build, Accessed 2019.

[15] Test Framework unittest. https://docs.python.org/2/library/unittest.html, Accessed 2019.

[16] Test Framework unittest2. https://pypi.python.org/pypi/unittest2, Accessed 2019.

[17] M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 447–450, 2017. URL https://doi.org/10.1109/MSR.2017.24.

[18] 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 '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pages 16–20, 2009. URL http://doi.acm.org/10.1145/1555860.1555866.

[19] V. Dallmeier and T. Zimmermann. Extraction of Bug Localization Benchmarks from History. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 433–436, 2007. URL http://doi.acm.org/10.1145/1321631.1321702.

[20] F. M. Delfim, S. Urli, M. de Almeida Maia, and M. Monperrus. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. To appear in SANER 2019.

[21] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405–435, 2005. URL https://doi.org/10.1007/s10664-005-3861-2.

[22] G. Gousios, A. Zaidman, M. D. Storey, and A. van Deursen. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 358–368, 2015. URL https://doi.org/10.1109/ICSE.2015.55.

[23] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.*, pages 191–200, 1994. URL http://portal.acm.org/citation.cfm?id=257734.257766.

[24] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440, 2014. URL http://doi.acm.org/10.1145/2610384.2628055.

[25] C. Le Goues, N. 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. URL https://doi.org/10.1109/TSE.2015.2454513.

[26] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for Evaluating Bug Detection Tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[27] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 643–653, 2014. URL http://doi.acm.org/10.1145/2635868.2635920.

[28] 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 *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 10–13, 2018. URL https://doi.org/10.1145/3196398.3196473.

[29] C. S. Timperley, S. Stepney, and C. Le Goues. BugZoo: A Platform for Studying Software Bugs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 446–447, 2018. URL http://doi.acm.org/10.1145/3183440.3195050.

[30] D. A. Tomassi. Bugs in the Wild: Examining the Effectiveness of Static Analyzers at Finding Real-World Bugs. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 980–982, 2018. URL https://doi.org/10.1145/3236024.3275439.

[31] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. How to Design a Program Repair Bot?: Insights from the Repairnator Project. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 95–104, 2018. URL https://doi.org/10.1145/3183519.3183540.

[32] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 805–816, 2015. URL https://doi.org/10.1145/2786805.2786850.