PExReport: Automatic Creation of Pruned Executable Cross-Project Failure Reports

Sunzhou Huang
Department of Computer Science
University of Texas at San Antonio
San Antonio, USA
sunzhou.huang@utsa.edu

Xiaoyin Wang
Department of Computer Science
University of Texas at San Antonio
San Antonio, USA
xiaoyin.wang@utsa.edu

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00027

***Abstract*—Modern software development extensively depends on existing libraries written by other developer teams from the same or a different organization. When a developer executes the software, the execution trace may go across the boundaries of multiple software products and create *cross-project failures (CPFs)*. Existing studies show that a stand-alone executable failure report may enable the most effective communication, but creating such a report is often challenging due to the complicated files and dependencies interactions in the software ecosystems. In this paper, to solve the CPF report trilemma, we developed PExReport, which automatically creates stand-alone executable CPF reports. PExReport leverages build tools to prune source code and dependencies, and further analyzes the build process to create a pruned build environment for reproducing the CPF. We performed an evaluation on 74 software project issues with 198 CPFs, and the evaluation results show that PExReport can create executable CPF reports for 184 out of 198 test failures in our dataset, with an average reduction of 72.97% on source classes and the classes in internal JARs.**

***Index Terms*—cross-project failure, executable failure report, failure reproduction, build tool, build environment, debloating**

I. Introduction

With the assistance of third-party infrastructures and functional components, software ecosystems have expanded tremendously over the past several decades. The prevalent usage of third-party software libraries has significantly reduced the cost of software development and improved the quality of software. As the size and complexity of software products increase, developers introduce more and more software dependencies via software libraries. However, the intricate dependencies among different software projects raise new maintenance challenges. When a developer executes the software, the execution trace often goes across the boundaries of multiple software products. So, for some execution failures, it may not be easy to determine which software project should take responsibility for and fix its code. In this paper, we refer to such failures as *cross-project failures (CPFs)*. For example, when a developer upgrades a software library and encounters a software failure, it may be caused by either a backward incompatibility bug in the software library or a non-robust usage of the library API.

This work is supported by NSF Grants CCF-1846467, CCF-2007718, and CSPECC-1736209.

Resolving CPFs usually requires the cooperation and negotiation of more than one project as opposed to intra-project failures (IPFs), so the failure needs to be reported from its observing software project to other projects. However, creating a good failure report is not trivial. A previous study on desirable bug reports [1] shows that a stand-alone executable test case is the most desirable information to be included in reports. A textual failure report from client developers may not be very helpful for the library developers to investigate or reproduce the failure. Since CPFs are typically triggered at the interface between the client code and the third-party software libraries, a thorough explanation of such issues in a failure report at least needs to involve some client-side software artifacts. However, in some situations, even if the client developers provide some code, the failure can also be hard to reproduce. For example, in the comment section of an Apache Issue (Issue 2497 in Apache TIKA project) [2], an Apache TIKA developer complained that "I'm not able to reproduce it w/in Solr in the unit tests with your file.". If the failure report contains a test case that can be compiled and executed in a stand-alone way, it would greatly simplify the diagnosis process.

An ideal failure report should satisfy three essential requirements: Executability, Readability, and Conciseness. Our pilot research reveals the existence of a trilemma when client developers try to create CPF reports using existing techniques. In particular, static and dynamic slicing techniques [3], [4], can both prune source code based on a given *seed code*, but they do not take into account the build process or the execution dependencies, so the generated slices may not be compiled or executed as a stand-alone project. Software debloating [5] is a practical technique for pruning software releases, but it focuses on destination code and execution-time dependencies instead of source code and compilation-time dependencies, so applying it to source code will cause the pruned code to be uncompilable. Finally, packaging the entire client software and sending it together as the CPF report is typically not a realistic solution, considering the numerous redundant dependencies that may (1) significantly increase the size of the report; (2) bring in lots of noise to the debugging process; (3) unintentionally leak internal information and proprietary code from other parties.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00027

In this paper, to achieve all three requirements, we developed the PExReport, a framework for generating executable pruned CPF reports. Given a CPF, PExReport performs a three-step analysis to trace the source code and object code (e.g., Java bytecode), which were loaded at compilation and run time. In the build process, it gradually identifies all the required source code and dependencies for compiling the tests and reproduces the CPFs. Besides source code and dependencies, PExReport further identifies the required portion of build configuration, resource files, and generated source code. Finally, PExReport extracts all identified required files from the original build environment and reconstructs a stand-alone project that can compile and reproduce the CPF.

We implemented PExReport for Maven [6] and Java combination, and performed an evaluation on 198 CPFs in 74 issues. Our results show that PExReport can reproduce 184 out of the 198 CPFs and achieve a pruning rate of 72.97% on source classes and the classes from internal JAR files.

To sum up, this paper makes the following contributions.

- A novel framework, PExReport, to extract both code and build dependencies and create pruned executable CPF reports.
- Three technical enhancements of PExReport to handle build configuration, resource files, and generated source code when performing automatic creation.
- An evaluation of PExReport on 198 cross-project failures from 74 software project issues, showing the effectiveness of PExReport on CPF reproduction and project pruning, as well as impacts of each enhancement.

II. Motivation

In this section, we conducted a pilot study in Java software ecosystems to provide real-world insights into good CPF reports and to illustrate the trilemma faced by current techniques, which motivates the design of our techniques.

A. Characteristics of Ideal CPF Reports

We conducted our pilot study using the JIRA issue tracker from Apache [7]. Upgrade incompatibility failures are strongly related to CPFs; therefore, we used three keywords (upgrade, incompatible, and Java) to retrieve 147 CPF reports from 32 Apache projects and tried to reproduce them.

Two researchers with more than five years of experience in software development were involved in this study. The first researcher recorded the CPF report and tried to reproduce the same failure in our build environment with the information from the report. If failure was not reproducible, the first researcher wrote down the reason, and the second researcher validated the reason. In cases where there was a conflict between the two researchers, a discussion was raised until all issues reached an agreement. Eventually, two researchers were only able to reproduce two of the 147 real-world CPF reports. We summarized the following reasons why these 145 CPF reports are difficult to reproduce; if a CPF report has more than one failed reason, we select the most direct one; the attached number is the frequency of reasons:

- **Never Reproduced (18)**. The client reporter provided an inaccurate test case, so the library developer never reproduced it either.
- **Environment Specific (43)**. The failure is environment specific, so researchers cannot reproduce it without knowing the environment settings.
- **Misuse (16)**. The client reporter misused the library, and the developer explained the reason.
- **No Test Code (41)**. The CPF report is pure textual, and researchers failed to create a test to reproduce the same behavior.
- **Fixed without Record (27)**. The failure was fixed without the specified fix commit, so researchers could not find the code version to reproduce it.

We believe these CPF reports do not deliver enough information for reproduction, especially for new developers who are not familiar with the software project's historical status. A previous study on desirable bug reports [1] also suggests a mismatch between what developers consider most helpful and what users provide. Based on these findings, we summarize the following characteristics of ideal CPF reports:

- **Executability.** A ideal CPF report should be easily reproduced by developers. An executable test case is the most desirable information to be included in reports. Considering the difference in build environments of client and library software projects, required code dependencies, configuration settings, and resource files should also be integrated into the test case.
- **Readability.** Keeping source code accessible is essential for developers to understand the issue. With a helpful code snippet, developers could accurately identify potential misuse to address CPFs for reporters. Due to the difficulty in locating the CPFs, a good report should retain all the related source code. Destination code is not equivalent to the source code for compiled languages. For example, Java bytecode is more readable than most other destination code, but its decompiled source code with the highest ranking decompiler retains only 78% of its semantics, according to a recent study [8].
- **Conciseness.** Considering a failure is typically triggered by a single execution, low redundancy is preferred in reporting scenarios, which could bring many benefits. A smaller report will reduce the time for network transmission and the space needed for storage (especially when there are many reports). Removing unnecessary code can significantly reduce the noise and accelerate the diagnosis progress. Furthermore, pruning redundant dependencies may also help avoid unintentionally sharing sensitive information or proprietary software artifacts.

B. Cross-Project Failure Report Trilemma

To further understand the obstacles in creating CPF reports, we explored existing automatic build tools and code pruning techniques. Figure 1 shows the CPF report trilemma in creating ideal CPF reports with current techniques, which may

partially explain why most reporters cannot create CPF reports effectively.

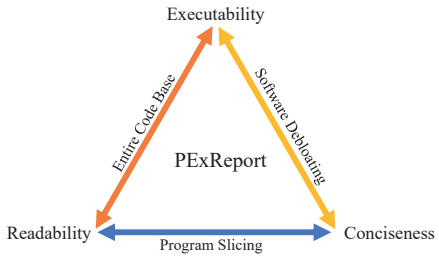

Fig. 1. Cross-Project Failure Report Trilemma

Modern automatic build tools. Maven [6] and Gradle [9] are popular tools for building and testing software in Java software ecosystems. Although both tools are designed to be platform-independent, we noticed that they lack features to extract a stand-alone test case. Using these tools can only provide an entire code base with public dependencies. Since these tools cannot identify unrelated dependencies, reporters only could attach the entire private dependencies to the test case. In brief, modern automatic tools only achieve executability and readability but do little to reduce redundant dependencies.

Program slicing. Program slicing [3] is a dataflow-based technique for pruning source code based on a specified *seed*, which is known to have limits owing to not working well with dynamic features. In addition, program slicing does not consider the execution environment or the build process. Therefore, the created slices of failures cannot be compiled or executed by themselves, which means it lacks executability.

Software debloating. In real-world ecosystems, debloating the size of applications is vital in embedded systems and application distribution. There are many practical tools and research studies [5] in this area. For example, ProGuard [10] is integrated into the Android build system, which only runs when building the application in release mode. It detects and removes unused classes, fields, methods, and attributes. In Java applications, released software does not contain source code. Because the output of compiled code is not equivalent to the source code, the developers will receive the CPF report with low readability. In short, the software debloat technique focuses on the software release stage, which does not consider the readability in the test stage.

None of the current techniques can perfectly solve this trilemma, which motivates the design of our approach to create a practical framework–PExReport, to focus on providing CPF reports with Executability, Readability, and Conciseness.

III. PExReport

In this section, we present PExReport, a framework designed to create pruned executable cross-project failure reports automatically. Figure 2 shows an overview of PExReport's

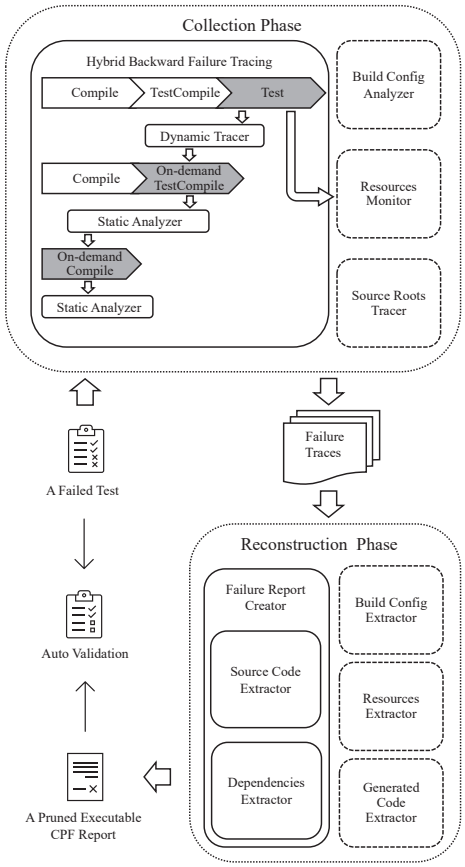

Fig. 2. Overview of the PExReport Workflow

workflow. PExReport is designed based on prevalent tasks of modern build tools involved in the lifecycle of CPFs. The input to PExReport is a failed test from an existing build environment, and the output is a pruned stand-alone executable CPF project. PExReport contains two major phases: (1) the collection phase to collect information about necessary source code, dependencies, and build environment by monitoring the underlying platforms: OS, the build tool (e.g., Maven), the compiler (e.g., Javac), and runtime (e.g., JVM), in the essential tasks of the modern build process; (2) the reconstruction phase to reconstruct a stand-alone build environment for the failed test based on the collected information. We name the collected information of a failed test from the first phase as failure traces. The arrows show the information flow between each component of PExReport. The reconstructed project for failure reporting will be automatically validated by checking whether the same error messages are triggered as in the original failure. Once the project is validated, it can serve as a reproduction package of the original failed test and will be reported as a pruned executable CPF report to the developers.

The *fundamental insight* of PExReport is that it first identified and addressed the problem of pruning the build and execution environment of a test failure on top of code dependencies. This is crucial for cross-project-failure reproduction because the environment is essential for a high reproduction

rate and cannot be easily shared. To overcome this problem, we develop Hybrid Backward Failure Tracing (Section III-C) which extracts and prunes the failure by monitoring the underlying platforms in essential build tasks (i.e., `Compile`, `TestCompile`, and `Test`) of the modern build process. It takes advantage of the fact that the underlying platforms already resolve all necessary dependencies in an on-demand build process and their resolutions are the most trustworthy for reproduction. Following the same insight, we further developed three novel enhancements (Section III-D) to support more heterogeneous build environment.

A. Principles of Design

PExReport should be easy to use and compatible with real-world projects and build ecosystems. We introduce the following principles to help reach the goals.

Source code preservation. When clients misuse the library, a snippet of source code can be extremely helpful. Developers could also easily trace the source code to identify the fault location. Considering that even the highest ranking decompiler does not always create semantically equivalent source code [8], source code preservation becomes more critical in CPF reports.

Build environment adaptation. A real-world project contains not only source code and dependencies; the build environment is also crucial. Modern build tools, such as Maven and Gradle, have been widely used by library developers. Given that PExReport requires CPF reports to be executable, it should be highly adaptable to modern build ecosystems.

Incremental creation of build environment. As the size and complexity of projects increase, modern build tools become complex and highly customizable. Identifying the redundant build environment could be inexhaustible and hard to scale. Therefore, PExReport opts for an incremental approach that creates the build environment by identifying necessary build dependencies, as opposed to a pruning method.

B. Prevalent Tasks for the Lifecycle of CPFs

The following tasks of modern build tools form the typical lifecycle of CPFs. These tasks are generally executed in the order below, but some tasks may be omitted if not required for the current project.

- **Generate sources (optional).** Generate additional source code for inclusion in compilation.
- **Process resources (optional).** Copy and process the resource files into the destination directory.
- **Compile (mandatory).** Compile the application source code of the project.
- **TestCompile (mandatory).** Compile the test source code, which is not coupled with the application source code.
- **Test (mandatory).** Run tests using a suitable testing framework; execute object code (e.g., bytecode) compiled from application and test source code.

The three mandatory tasks, namely `Compile`, `TestCompile` and `Test`, are the basic steps used by build tools to reproduce a test failure. The main reason to use both `Compile` and `TestCompile` to compile source code is to make the application source code independent of the test source code. Code generation and resource management are quite popularly used in complex real-world applications. Although they are optional tasks, we should not underestimate them in reproducing real-world CPFs.

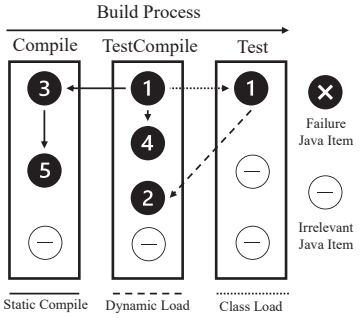

Fig. 3. Exemplar Three Tasks Build Process of Java Project

C. Base Approach: Hybrid Backward Failure Tracing

In the collection phase, our base component is hybrid backward failure tracing, a three-step analysis that traces failure-related source code and library dependencies. This component compiles and executes the failed test in its original build environment, and tracks items at `Test`, `TestCompile` and `Compile` tasks, which are essential for any failure source code to be compiled and executed.

Figure 3 shows the dependency tree in a build process for a failed test of Java project with three basic tasks. We refer to source code and object code (bytecode) as *items*, which are used to compile and execute the failed test, respectively. All the circles are the items of the Java project, a black circle means a failure item, and a white circle means an irrelevant item. The dotted line arrow is for class loading, so the two *Item 1* in the two tasks are equivalent but in a different form (source code in `TestCompile` task, bytecode in `Test` task). The solid line arrows show the static dependencies, and the dashed line arrows show the dynamic dependencies. Following the build process order, the compiled application source code in `Compile` task is provided to the subsequent `TestCompile` task as dependencies, after that, the compiled test source code of `TestCompile` is loaded to `Test` task for execution. For example, the `Compile` task compiles *Item 3* (application source code), which is provided to `TestCompile` task as a reference later; the *Item 1* (bytecode) of `TestCompile` task is loaded to `Test` task. The build process must be irreversible to ensure that the previous item will not depend on the later item. Given that the build process cannot predict the required items for the subsequent task, PExReport must perform the build process in three rounds to track all failure items. For example, in Figure 3, the reference $1 \rightarrow 2$ can only be discovered in `Test` task, reference $1 \rightarrow 3$ can only be discovered in `TestCompile` task, and the reference $3 \rightarrow 5$ can only be discovered in `Compile` task. If a CPF occurs, as failures happen at the end of the build process (`Test` task), a backward tracing will be performed to obtain the failure traces.

The details of Hybrid Backward Failure Tracing are described in Algorithm 1.

Algorithm 1 Hybrid Backward Failure Tracing Algorithm

Input: Failed test t, all test source code C_t, all application source code C_s, all library object code O_l

Output: $Traces$

1: Initialize trace of C_t: Set $T \leftarrow \emptyset$
2: Initialize trace of C_s: Set $S \leftarrow \emptyset$
3: Initialize trace of O_l: Set $L \leftarrow \emptyset$
4: **Round 1:**
5: $Compile$: Object code $O_s \leftarrow$ Compile C_s by referencing O_l;
6: $TestCompile$: Object code $O_t \leftarrow$ Compile C_t by referencing O_s and O_l;
7: $Test(t)$: Record $R_1 \leftarrow$ Execute t by dynamic loading O_s, O_t and O_l;
8: $T, S, L \leftarrow DynamicTracer(R_1)$;
9: **Round 2:**
10: $Compile$: reuse O_s;
11: $TestCompile(T)$: Record $R_2 \leftarrow$ On-demand compile test source code in T by referencing O_s and O_l;
12: $T, S, L \leftarrow StaticAnalyzer(R_2)$;
13: **Round 3:**
14: $Compile(S)$: Record $R_3 \leftarrow$ On-demand compile application source code in S by referencing O_l;
15: $S, L \leftarrow StaticAnalyzer(R_3)$;
16: **return** $Traces\{T, S, L\}$

In **Round 1**, PExReport runs the whole build process to the last task–`Test` with entire code base and library dependencies for both `Compile` and `TestCompile` tasks, and executes only the target failed tests in the `Test` task. PExReport uses the log of build tool to record all dynamically loaded items in the `Test` task. These items are required for the execution of the failed test in run-time, so we refer to them as R_1. For example, in a Java project, R_1 are the bytecode (object code) loaded into JVM, also referred to as Java class. The $DynamicTracer$ analyzes R_1, and collects the corresponding traces (usage of items).

In **Round 2**, PExReport runs the build process up to the `TestCompile` task, reusing the object code compiled from all application source code in `Compile` task. For the `TestCompile` task, PExReport on-demand compiles test source code collected from *Round 1* to avoid compiling irrelevant items. In `TestCompile` task, PExReport records all referred items as R_2. These items are required for the compilation of test source code, so $StaticAnalyzer$ collects and combines the corresponding traces.

In **Round 3**, PExReport runs a single `Compile` task of the build process, trying to compile only the application source code collected from *Round 1* and *Round 2*. In the `Compile` task, PExReport records all referred items as R_3. These items are required for the compilation of failure related application source code, so $StaticAnalyzer$ collects the corresponding traces and combines them with previous traces.

Traces. PExReport returns the collected failure traces of test source code, application source code and library object code in Algorithm 1, and sends the traces to the failure report creation component to be included in the executable CPF reports.

PExReport uses the build tools to resolve dependencies to increase robustness and non-invasiveness. Modern build tools can fetch information directly from compilers or virtual machines, meaning the resolved dependencies are the most trustworthy. The inferred dependencies from the analysis tools may be incomplete due to the quality of the tools, and some invasive tools can change the behavior of the analyzed projects. However, the build tools only provide the resolved dependencies without the dependency tree, which means that PExReport cannot use all the items in each task that may contain irrelevant items during the build process. As shown in Algorithm 1, to solve this problem, PExReport uses on-demand compilation[1] to compile only the failure related source code and then use the references of compilation as failure traces.

The dynamic loading and static reference are handled by the run-time environment and the compiler, respectively. PExReport uses the dynamic tracer to monitor dynamic item loading in the `Test` task and the static analyzer to monitor the reference in the `Compile` and `TestCompile` tasks due to the fact that the information provided by the build tools differs between compilation and execution. The current PExReport supports the default Java compiler and multiple customized compilers, such as *javac-with-errorprone*; if any compiler is not supported, PExReport only needs an adjustment to the static analyzer. As the build tools sort different types of items into different directories, the PExReport can accurately categorize items with the location information and further allow the Failure Report Creator to place them in corresponding directories.

D. Three Enhancements

As discussed in Section II, besides source code and library dependencies, the build environment also plays an important role in the reproduction of failed tests. Although the base approach can still reproduce the failure with source code and library dependencies in the minimal standard build environment, in order to enhance the reproduction rate for real-world complex projects, we developed the following enhancements over the base components to further reconstruct a reliable build environment:

- **Handling of the build configuration.** In the collection phase, the build configuration analyzer fetches the resolved build configuration and determines the necessary values of them, which will be then inserted into the template project by the build configuration extractor in the reconstruction phase.

[1]On-demand compilation, such as the implicit compilation in Java, which allows the compiler to search the required source code and dependencies to compile the designated source code.

- **Handling of resource files.** For the collection phase, we developed the resource monitor to watch all file accesses in the project directory during the test execution at the operating system level. Then, in the reconstruction phase, our resource extractor will copy and insert pruned resource files into the proper locations of the reconstructed project.
- **Handling of source code generation.** Our source roots tracer tracks source code generation in the collection phase, and the generated code extractor locates the generated source code and extracts the needed source code to skip unnecessary code generation and avoid code generation conflicts.

Handling build configuration. Build configuration values are necessary parts of an executable failure report. Including unnecessary configuration values will lead to more noise in the debugging phase (the debugging developer needs to consider more factors), more information leaks (e.g., internal emails and file paths), as well as potential build failures, for example, the configuration values may refer to pruned source code and dependencies which no longer exist. Therefore, we enhance PExReport to further identify and extract only build configuration values necessary for the failed test reproduction.

In the Collection Phase, the Build Configuration Analyzer fetches the *effective build configuration* (e.g., Maven provides *help:effective-pom* [11]), which the build tools use to build the failed test at run time. The *effective build configuration* gathers all build configuration values scattered in all build configuration files, and resolves configuration value overwriting among multiple configuration files based on the build configuration hierarchy and resolution rules, for example, Maven picks the "nearest definition" for resolving dependencies. However, the *effective build configuration* is still redundant for reproducing the failed test, compared with the required configuration values of the three main build phases (compile, test-compile, and test). Since tasks in the build process are performed by various plug-ins and the plug-ins can be attached to different build phases (e.g., the Java compiler plug-in can be attached to the compile and test-compile phases), PExReport further leverages the attachment relationship to identify all the plug-ins that are attached to the three basic tasks. It also excludes code-style checking and analysis plug-ins because they do not directly affect the compilation or testing. After all necessary plug-ins are identified, the Build Configuration Extractor extracts the configuration of the plug-ins from the *effective build configuration* by querying with XPath. After modifying the static information, such as the absolute project path, the extractor feeds all the extracted information to the build configuration file of the reconstructed project.

Handling resource files. The Resource Monitor identifies necessary resource files for the failed test reproduction by using file system monitoring API (e.g., *inotify* [12]–a Linux kernel subsystem that monitors filesystem events) to watch the resource file access during the `Test` phase. We use an example in Figure 4 to illustrate how we monitor and extract resource files, the left side is an original resource folder

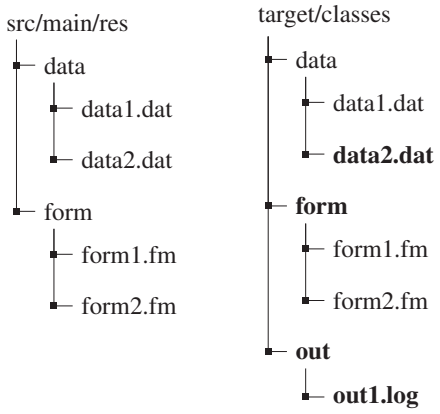

Fig. 4. Exemplar Resource Directory Structure

existing before the build, and the right side is a target resource folder generated after the build. The files/folders accessed during testing are highlighted in bold font.

As shown in Figure 2, the resource monitor only outputs files accessed at `Test` task because the `Process resources` task often copies all resource files to the target folder (e.g., the whole directory structure on the left of Figure 4). For necessary resource files accessed during the compilation process (e.g., templates of generated source code), we specially handled them by the Generated Code Extractor. Our resource monitor also ignores all files/directories generated during the build/test process (e.g., folder `out` and file `out1.log` in Figure 4, *.class* files) because these files should not be included in the reproduction package (unnecessary and causing path conflicts). For the files copied to the target location from source locations (e.g., folders `data` and `form` and all their files in Figure 4), we do not consider them as generated files, because the Resource Monitor can trace back to their source copies in the original project.

During the reconstruction phase, our Resource Extractor extracts the files and directory structure from the original project based on information collected by the Resource Monitor. For copied resource files, the Resource Extractor uses their source copies and paths tracked by the Resource Monitor (e.g., extracting `data2.dat` from the original resource folder because its copy in the target folder is accessed). Note that maintaining the structure of an accessed directory is also crucial for test reproduction. The failed test may access the directory but never access the files under it (e.g., checking the existence of a file), and some tests require a special directory structure to reproduce successfully. The Resources Extractor creates empty dummy files for the un-accessed files under the accessed directory to retain the directory structure and cover this situation. For example, in Figure 4, folder `form` is accessed and can be traced back to the original resource folder, but none of its files is, so the folder will be extracted, and all its files will be replaced with empty files with same names.

Handling source roots and generated code. The software build process may generate new source code in various ways,

such as creating code from template files, generating parsing code from syntax/XML files, and even directly fetching source code from remote locations. Furthermore, code generation is often implemented in third-party tools and plugins. To handle such high flexibility of code generation in a general way, we enhance PExReport by omitting the code generation process and directly including the generated source code in the test reproduction package.

At `Generate sources` task, the build tools use source code root paths (Source Roots) to locate all (original and generated) source code. In the Collection Phase, our Source Roots Tracer tracks all the accessed source code root paths from the debug information of compilation. Next, the Generated Code Extractor utilizes the paths to identify the generated source code and excludes all original source code. In addition, the build tools may also generate some source code from code annotation processing. Our Generated Code Extractor excludes such code because it will cause compilation conflicts.

E. Automatic Creation of CPF Reports

In the reconstruction phase, our base component is failure report creation. This component creates a template project with a standard build configuration for the failure report and adds the required source code and dependencies into the project.

The Failure Report Creator uses a customizable template to generate a standard project structure for the reproduction package. It uses different extractors to fetch required source code, dependencies, resource files, and build configuration. As two basic components shown in Figure 2, the Source Code Extractor converts the required item name to code file paths and copies these files to the generated reproduce project while maintaining the original structure; the Dependencies Extractor duplicates the local and remote dependencies and reduces the unnecessary portion based on failure traces. The generated project organizes and links the required source code and dependencies in a standard build configuration file for reproduction. As mentioned earlier, in many cases, we also need to extract the build environment accordingly so that the replication package can successfully reproduce the failure. Therefore, the Failure Report Creator can further extract the required build configuration, resource files, and generated code by incorporating the Build Configuration Extractor, Resources Extractor, and Generated Code Extractor, respectively.

F. Failure Report Validation

After the final executable failure report (i.e., reproduction package) is constructed, the Report Validator uses a conservative strategy to ensure the report can reproduce the original test failure. In particular, a report is validated only if it generates the identical failure type and message after executing the test. Note that this strategy may reject some successfully reproduced test failures (e.g., when failure messages change with date, time, or absolute path), but it allows developers to trust the pruned failure report (and our evaluation to be conservative). Note that in practice the reporter could still manually validate the report when the automatic validation fails.

IV. Evaluation

This section presents our experimental results by answering the following research questions:

- **RQ1: How effective is PExReport in creating executable CPF reports? (Executability)** To answer this research question, we counted the number of subjects that PExReport can exactly reproduce with the same failure type and message, and calculated the reproduction rate.
- **RQ2: How does PExReport perform on project pruning? (Conciseness)** To answer this research question, we calculate the reduction rates of different items from subjects and present cumulative graphs to show the performance of pruning. If a reproduction fails, we use the entire project as the CPF report (0% reduction rate).
- **RQ3: How effective are our techniques in PExReport on solving the CPF report trilemma?** To answer this research question, we performed an ablation analysis using reproduction and reduction rates.

A. Experiment Setup

We implemented PExReport in Python, based on Apache Maven [6] build tool. Since Maven is used primarily for Java, we chose Java projects as our target projects. The Archetype [13], a Maven project template toolkit, is used for generating a standard Maven project for reproduction.

We performed our experiment on a Linux server running Ubuntu 16.04.5 LTS with two 8-core 2.6 GHz CPUs and 512 GB RAM. Apache Maven 3.6.3 was used to build and run tests. To avoid race conditions, we solely executed each build and test on the server. An automatic Failure Validator examined the failure types and messages to ensure the failures had been successfully reproduced. If a failed reproduction was reported, the entire project was provided as a failure report for evaluation.

B. Metrics

To answer **RQ2**, we need to use some general metrics to compare pruning performance among all subjects. Well-defined metrics could also help us to understand the results correctly. PExReport prunes the entire building environment of the failed test, including source code, dependencies, build configuration, and resource files. So, our metrics should measure the pruning on all of them and we define the following metrics:

- **# Internal classes**: The number of classes from the JAR dependencies within the same organization of the subject. In the Maven convention, the same organization typically shares the same domain name in their group ID of libraries. So, we compare the group IDs to identify internal libraries and count the internal classes inside libraries.
- **# Source classes**: the number of classes compiled from the Java source code.

- **# Source+Internal classes**: The total number of source and internal classes. Our subjects have vastly different distribution strategies for main classes and internal classes. So, we combine these classes to provide a more generic metric and reduce the interference of different code distribution strategies. Since we measured the total of source and internal classes, and classes could be compressed in JARs, we chose the number of classes instead of the size of classes.
- **Size of build configuration**: The total number of characters from the loaded POMs (Maven build configuration files), excluding the whitespace and *dependencies* section for a fair comparison. We uses *# internal classes* as a metric for dependencies, which is more precise.
- **# Resources**: The number of resource files within the resource directories of the project.
- **Percentage of reduction**: The percentage of reduced items, defines as the ratio of removed items to original items. For example, the percentage of reduction of internal classes is the ratio of the number of pruned internal classes to the total number of internal classes in original projects.

C. Dataset Construction

Our experiment dataset is constructed from the benchmark dataset of a research project–Sensor [14], which triggers failures by changing dependency versions of open-source Java projects. The Sensor ground truth dataset contains 316 semantic conflicts confirmed by researchers. In our dataset, we refer to one unique failed project-library pair as one cross-project issue and one failed test case as one CPF. Although PExReport can handle CPFs raised by multiple test cases (treat multiple tests as one test), clustering tests based on dependencies is beyond the scope. Given that each issue may contain multiple CPFs, our experiment dataset consists of 74 issues with 198 CPFs.

We do not use every CPF in Sensor dataset as subjects to evaluate PExReport because (1) some issues are irreproducible due to environmental change; (2) some CPFs are irreproducible solely since they are dependent on other CPFs (clustering non-independent tests is beyond the scope of this paper); (3) some CPFs have random factors or are flaky, so they may cause uncertainties during validation; and (4) many CPFs in one issue are identical, so using all of these repetitive failures will dominate the results. Therefore, we performed the selection for representative CPFs in the following steps:

- **Step 1: Verifying failed issues.** After removing duplicated project-library pairs, we downloaded projects into our server and executed them without applying any changes. For those successfully executed projects, we re-executed them with conflicting libraries based on the Sensor dataset. Then, we selected all the issues that only failed with changed dependencies in our experiment environment. We set a 900-second time limit for each execution and removed all projects with timeout and compilation failures. Only three issues were discarded due to timeout; 119 issues were collected throughout this step.
- **Step 2: Clustering CPFs.** We executed all CPFs three times and removed all CPFs with inconsistent output over three executions, after which five issues were discarded. For each of the remaining issues, we clustered CPFs with the identical failure type and message (failure group) into one cluster. As a result, we clustered 6,415 CPFs into 1,020 failure groups from 114 issues. The maximal number of failure groups within an issue is 367.
- **Step 3: Selecting representative CPFs as subjects.** We observed that an issue may have significantly more failure groups (i.e., 367) than others because test failures whose messages contain unique numbers (e.g., Java Object ID) are clustered into distinct failure groups. In addition, ninety percent of failure types contain no more than four failure groups. To avoid over-representation, we further considered failure types (e.g., Assertion Failure, No Such Method) by selecting up to four failure groups at random for each failure type and issue, and then randomly selecting one representative CPF for each selected failure group. In this step, we only selected CPFs that are solely reproducible; 105 out of 395 (26.5%) failure groups and 45 out of 238 (18.9%) failure types have been discarded due to none of their CPFs being solely reproducible. We observed that 33 of the 107 selected issues lack internal libraries. For a more comprehensive and consistent evaluation, we only use the 74 issues with internal libraries.[2]

TABLE I
THE STATISTICS OF THE COLLECTED ISSUES.

Item	Min.	Max.	Mean	Med.	$\neq 0$
# Internal classes	32	36,389	3,848	955	74
# Source classes	8	2,457	265	144	74
# Source+Internal cls	90	36,417	4,113	1,331	74
Size of Config	8,428	77,735	25,349	21,746	74
# Resources	0	655	55	8	68
# Total issues					74

Eventually, we collected 198 representative CPFs from 74 issues. Table I shows the statistics of the issues in our dataset. The symbol $\neq 0$ means not equal to zero. The $\neq 0$ column shows the count of issues that have at least one corresponding item. For instance, an issue that does not have any resource files will not be counted.. The statistics show that the issues in our final dataset have a large number of classes (4,113 Mean and 1,331 Median of *# Source+Internal classes*).

We further categorized the 198 CPFs based on their failure type to show the diversity of failures in our dataset. The total number of different failure types is 31, and the top 10 failure types are shown in Figure 5. The top two failure types are assertion-related, representing more than half of CPFs.

[2]The data used in this paper and PExReport implementation are available at https://doi.org/10.5281/zenodo.7578677; additional evaluation results for the 33 issues and concrete examples can be found on our website: https://sites.google.com/view/PExReport/home

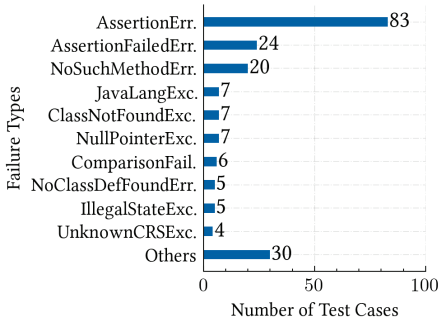

Fig. 5. Top 10 Failure Types in Representative CPFs

The assertion-related errors indicate that the actual variable values are not equal to the expected values. Another common failure type is "not found". If the dependency change modified class/method signatures, the program could still fail in the `Test` task in the case of reflection.

D. Evaluation Results

1) The answer to RQ1:

To answer the RQ1, we executed PExReport on 198 representative CPFs and summarized the results in Table II. PExReport successfully reproduced 184 out of 198 CPFs with exact failure types and messages, and achieved a high reproduction rate of 92.93%.

TABLE II
REPRODUCED CPFS

Implementation	# Repro. CPFs	Repro. Rate	Perf.(sec)
PExReport	**184**	**92.93%**	**73.03**
w/o Dynamic	48	24.24%	66.51
Source & Deps.	46	23.23%	51.22
w/o Build Config	101	51.01%	65.79
w/o Resources	104	52.53%	68.61
w/o Generated Code	146	73.74%	55.08
# Total CPFs	198		

We investigated the 14 unreproduced CPFs to understand why PExReport cannot reproduce them. 9 of the 14 CPFs failed because of the unsupported compilers (i.e., Groovy [15] compiler), which could not provide the class reference information for Hybrid Backward Failure Tracing. The rest of the 5 unreproduced test cases are missing required classes at run time. We believe the Java Runtime failed to provide all the information on required classes during the test execution. The mistracking could happen when a program checks the existence of a class but never access it.

2) The answer to RQ2:

We further calculated the percentage of reduction for each metric and presented Table III to show the statistics of the percentage of reduction. PExReport performed a 55.37% of average reduction rate on required source classes and outperformed on internal classes, source+internal classes, build configuration, and resources with 75.94%, 72.97%, 82.96%, and 74.18%, respectively.

TABLE III
THE STATISTICS OF REDUCTION RATE

Percent Reduction	Min.	Max.	Mean	Med.
# Internal classes	0.00%	100.00%	75.94%	84.67%
# Source classes	0.00%	99.05%	55.37%	64.39%
# Source+Internal cls	0.00%	99.65%	72.97%	79.95%
Size of Config	0.00%	96.76%	82.96%	89.02%
# Resources	0.00%	100.00%	74.18%	90.91%

We assume the reporter still wants to use the entire project as a failure report if PExReport fails to reproduce the failure. Therefore, the percentage of reduction is 0% for unreproduced subjects. We presented the cumulative plot of results in Figure 6. The S+I represents Source+Internal, shown as the orange curve. The y-axis is the fraction of x that has satisfied the $\geq$ condition, which means that the fraction of results have at least a certain percentage of reduction. For example, a point ($x = 60\%$, $y = 0.79$) on the blue curve (Internal) shows that 79% of CPFs have a reduction rate over 60% on internal classes.

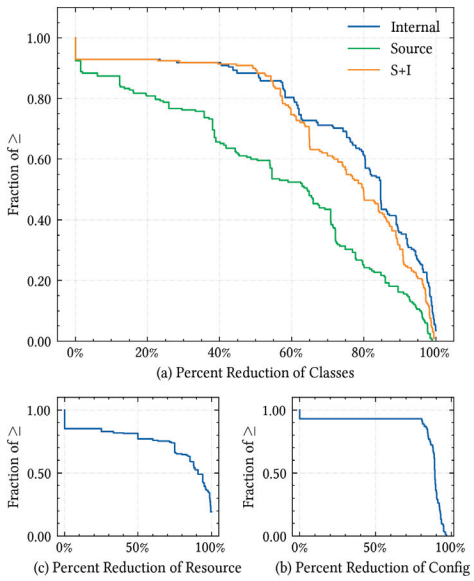

Fig. 6. Cumulative Step Graph of Reduction Rates

PExReport performed impressively on almost all the metrics. As the source code can provide excellent readability to developers, we believe 55.37% of the average reduction rate is acceptable in the CPF report. Our investigation shows that the high reduction rate for build configuration is based on non-test-related configuration, such as the deployment settings and unused plugin settings. Figure 6 shows that PExReport could provide a stable reduction rate for the redundant dependencies. In conclusion, the high reduction rates indicate that PExReport could provide conciseness to CPF reports.

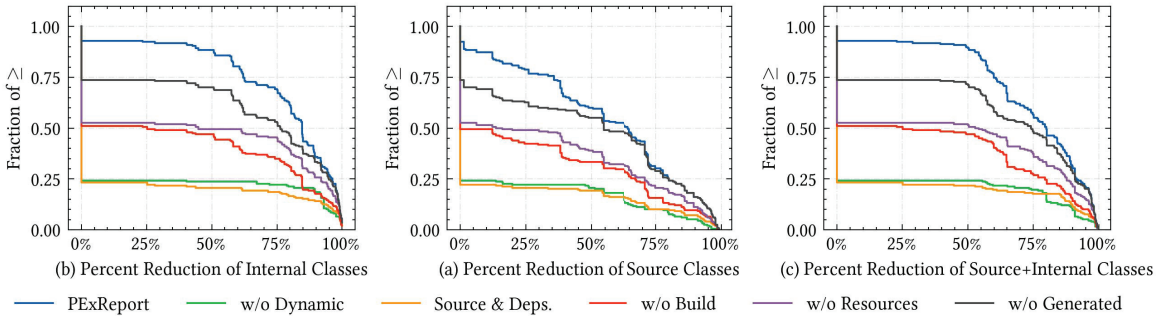

Fig. 7. Comparison of Different PExReport Techniques

3) The answer to RQ3:
Besides the Hybrid Backward Failure Tracing, PExReport applies three enhancements to handle build configurations, resource files and generated code. We performed an ablation analysis on PExReport and calculated reproduction and reduction rates.

w/o Dynamic represents PExReport without the dynamic tracer (JVM monitoring), which still retains the static analyzer and three enhancements. As shown in Table II, *w/o Dynamic* only achieved a low reproduction rate of 24.24% by reproducing 48 of 198 CPFs. Compared with the high reproduction rate of 92.93% from PExReport, it states that pure static analysis could perform terribly in real-world Java applications as large Java applications use various dynamic features. *Source & Deps.* represents PExReport without three enhancements, which provides source code and dependency packages with the standard Maven build configuration. The *Source & Deps.* only reached 23.23% (46 of 198 CPFs), a low reproduction rate. Although source code and dependencies contain the most information that developers expect the reporter to provide, it is still highly likely that developers cannot use them with the standard build setting to reproduce the CPF, as exemplified in Section II. To better understand the effectiveness of the three enhancements, we turned off each enhancement one by one and compared the results with the integrated PExReport. As shown in Table II, the impacts of these enhancements are large. The reproduction rate reduces from 92.93% to 51.01% and 52.53% without enhancements on build configuration and resource files, respectively. Without the generated code enhancement, 73.74% of CPFs can still be reproduced because code generation is not a must for many Java projects. In contrast, developers need to deal with build configuration in every project, so the corresponding enhancement has the largest impact.

We also presented the reduction results of ablation analysis in Figure 7, using cumulative plots to intuitively compare our techniques. As shown in the performance column of Table II, PExReport's execution time is not heavily affected by these enhancements. As creating CPFs is not a repeated activity like the standard build process, the average reproduction finished in a minute or so should be acceptable.

Overall, PExReport reproduced the majority (92.93%) CPFs and provided executable cross-project failure reports with an average of 72.97%, 82.96%, and 74.18% reduction rate for source and internal classes, build configuration, and resources, respectively.

E. Threats to Validity

The major internal threat to validity is the potential errors in our scripts and tool building. To reduce this threat, we carefully checked the implementation and shared them for peer review. The major external threat to validity is the variance of projects and test failures that may not be covered by our evaluation. To reduce this threat, we constructed our dataset based on the ground truth research dataset–Sensor [14] and carefully chose CPFs to avoid bias. Although PExReport is designed to work on all reproducible failures, our evaluation does not consider flaky tests and non-independent tests to avoid uncertainties and inconsistencies. Consistently reproducing and clustering failures are beyond the scope of this paper, but may also require future research.

V. Discussion

Generalization to Other Languages and Building Tools. PExReport is designed based on Maven and implemented for Java programming language. However, the idea of our approach is general and may be applied to other programming languages and build tools. The dependency analysis and enhancements for build configuration, resource files, and generated code are generalizable to most JVM languages, but compiler integration may be different from language to language, so the Hybrid Backward Failure Tracing component needs to be adjusted to support new languages. Other build tools such as Gradle / Make may allow more complicated file operations so more advanced analysis of the build process may be required.

Duplicate Failure Reports. PExReport takes a single CPF as its input, but one issue may generate more than one CPFs. PExReport does not resolve the potential duplication of CPFs but it can be resolved using test failure clustering and selection of representative CPF from each cluster. In our experiment, such clustering largely reduced the client CPFs to be considered. In practice, the developer may manually

determine which CPF should be reported and which one should not, and select a representative CPF. Furthermore, even if the developer chose two CPFs with a similar root cause, after the pruning, representative CPFs sharing the same root cause may be reduced to similar executable CPF reports. In such cases, the client developer may need to double-check the similarity between executable CPF reports before submitting them.

VI. Related Work

Though we are not aware of any research efforts creating pruned executable CPF reports to solve the CPF report trilemma, our techniques are related to existing efforts on bug reproduction, program slicing, software de-bloating, and fault localization.

Bug and Crash Reproduction. Recently, JCHARMING [16] uses crash traces and model checking to identify program statements needed to reproduce a crash. Liu et al. proposed DoubleTake [17], which uses evidence-based analysis to largely reduce the cost of recording erroneous states. Huang and Zhang proposed LEAN [18] to reduce the complexity of the replayed trace and the length of the replay time without losing the determinism in reproducing concurrency bugs. Weeratunge et al. [19] propose a novel approach that performs a lightweight analysis of a failing execution in a multi-core environment and reproduces the bug in a single-core system, under the control of a deterministic scheduler. Herbold et al. [20] proposed a generic and non-intrusive GUI usage monitoring mechanism to record and replay GUI bugs. Moran et al. [21] proposed a technique that records user action steps when reproducing an Android bug, and automatically fills them into a bug report. Some other research efforts [22] [23] try to automatically construct build or execution environments but they have not been applied to bug reproduction. Compared with existing approaches in this area, our approach focuses on the pruning and reconstruction of the build environment for buggy execution, including build configuration, resource files, and generated code.

Program Slicing. Program slicing [3] is a technique to carve from a large program a smaller program that implements one or multiple features of the larger program. Program slicing often relies on code dependency graph [24] [25] and has been used to prune a lot of targets from source code [26] to pre/post conditions [27], paths [28], and databases [29]. Dynamic slicing [4] [30] identifies the statements that the buggy output depends on. Executable union slicing preserves the meaning of the original program using conditioned slicing. [31] Compared with program slicing, PExReport focuses more on the build environment. The low reproduction rate of PExReport's variant without enhancements shows that fetching only code dependency is not sufficient. On the other hand, PExReport's enhancements can also be viewed as a slicing of the build environment, including the build configuration and the resource files.

Software De-bloating. Software de-bloating techniques prune a released software package to remove unnecessary dependencies and thus reduce its size, loading time, and attack surface. Pure static de-bloating techniques such as ProGuard [10] and Jax [32] rely on static program slicing and more recent work such as JShrink [5] further consider runtime dependencies. Although both pruning code, compared with PExReport, the goal of de-bloating is to retain software features instead of reproducing a single execution. Also, although de-bloating may generate executable pruned code by tracking execution dependencies, it does not work on source code and does not retain the build environment.

Fault Localization. Statistical fault localization [33] [34] gives a suspicious score to each statement (or other types of code structures such as sub-control-flow-graphs [35]) according to the number of successful and failed test cases that cover the statement. IR-based fault-localization [36] evaluate suspiciousness scores of statements by their textual similarity to the descriptions in a given bug report. Change-aware fault localization, such as Delta-debugging [37] [38] [39] considers the scenario of localizing regression faults when the history between the successful version and failed version is available. Other approaches [40] [41] perform impact analysis on code changes to localize faults. Hassan and Wang [42] studied localization and repair of bugs in build scripts. On cross-project bugs, Mostafa et al. [43] and Chen et al. [44] studied behavior backward incompatibilities and their detection. Compared with all these techniques, PExReport focuses on reproduction instead of localization of bugs, so it needs to further identify and package all dependencies of a bug in its build and execution process.

VII. Future Work

In the future, we plan to enhance our research in the following directions. First of all, we plan to enlarge our dataset to evaluate PExReport on more CPFs in more projects. Second, we plan to further prune the created executable CPF reports with finer-grained analysis and perform source code detailed reduction. Third, we plan to perform user studies to understand how much our tool can help developers in reproducing real-world bugs. Fourth, we plan to expand PExReport with other build tools by developing more advanced features for different software ecosystems.

VIII. Conclusions

An executable test case is one of the most desirable features of failure reports. When reporting *cross-project failures (CPFs)* to library developers, a test case is even more helpful because code is a natural way to describe interactions between library code and client code. In this paper, we developed PExReport, a framework to automatically create pruned executable CPF reports for developers, and solve the CPF report trilemma. PExReport uses Hybrid Backward Failure Tracing to identify the necessary source and dependencies, and has further enhancements to handle build configurations, resource files, and generated code. Our evaluation shows that PExReport can produce pruned executable CPF reports for 184 of 198 CPFs with an average reduction rate of 72.97%.

REFERENCES

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 308–318.

[2] T. Allison, "Tika/tika-2497," https://issues.apache.org/jira/browse/TIKA-2497, 2017.

[3] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[4] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990, pp. 246–256.

[5] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "Jshrink: In-depth investigation into debloating modern java applications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 135–146.

[6] "Apache maven," https://maven.apache.org/, 2002.

[7] A. S. Foundation, "Apache's jira issue tracker," https://issues.apache.org/jira/secure/Dashboard.jspa, 1999.

[8] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *Journal of Systems and Software*, vol. 168, p. 110645, 2020.

[9] "Gradle," https://gradle.org/, 2008.

[10] "Proguard: Java and android apps optimizer," https://www.guardsquare.com/en/products/proguard, 2022-08-01.

[11] "Apache maven help:effective-pom," https://maven.apache.org/plugins/maven-help-plugin/effective-pom-mojo.html, 2022.

[12] "inotify," https://man7.org/linux/man-pages/man7/inotify.7.html, 2005.

[13] "Apache maven archetype," https://maven.apache.org/guides/introduction/introduction-to-archetypes.html, 2002.

[14] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z.-l. Zhu, "Will dependency conflicts affect my program's semantics," *IEEE Transactions on Software Engineering*, 2021.

[15] "Apache groovy," https://groovy-lang.org/, 2003.

[16] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "Jcharming: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 101–110.

[17] T. Liu, C. Curtsinger, and E. D. Berger, "Doubletake: Fast and precise error detection via evidence-based dynamic analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 911–922.

[18] J. Huang and C. Zhang, "Lean: Simplifying concurrency bug reproduction via replay-supported execution reduction," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 451–466.

[19] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing multicore dumps to facilitate concurrency bug reproduction," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 155–166.

[20] S. Herbold, J. Grabowski, S. Waack, and U. Bünting, "Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 232–241.

[21] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 673–686.

[22] F. Hassan, R. Rodriguez, and X. Wang, "Rudsea: recommending updates of dockerfiles via software environment analysis," in *Proceedings of the 33rd acm/ieee international conference on automated software engineering*, 2018, pp. 796–801.

[23] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 38–47.

[24] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow, "Cluster analysis of java dependency graphs," in *Proceedings of the 4th ACM symposium on Software visualization*, 2008, pp. 91–94.

[25] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 457–466.

[26] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 112–122.

[27] M. Harman and R. Hierons, "An overview of program slicing," *software focus*, vol. 2, no. 3, pp. 85–92, 2001.

[28] R. Jhala and R. Majumdar, "Path slicing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language Design and Implementation*, 2005, pp. 38–47.

[29] D. Willmor, S. M. Embury, and J. Shao, "Program slicing in the presence of database state," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 448–452.

[30] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003, pp. 319–329.

[31] S. Danicic, A. De Lucia, and M. Harman, "Building executable union slices using conditioned slicing," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, 2004, pp. 89–97.

[32] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for java," in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1999, pp. 292–305.

[33] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02, 2002, pp. 467–477.

[34] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, 2005, pp. 273–282.

[35] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09, 2009, pp. 141–152.

[36] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 1–11.

[37] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 1999, pp. 253–267.

[38] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, pp. 183–200, 2002.

[39] K. Yu, M. Lin, J. Chen, and X. Zhang, "Practical isolation of failure-inducing changes for debugging regression faults," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, Sept 2012, pp. 20–29.

[40] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, 2011, pp. 23–32.

[41] X. Ren and B. G. Ryder, "Heuristic ranking of java program edits for fault localization," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007, pp. 239–249.

[42] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1078–1089.

[43] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: a study on behavioral backward incompatibilities of java software libraries," in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, 2017, pp. 215–225.

[44] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 112–124.