History-Driven Build Failure Fixing: How Far Are We?

Yiling Lou†
Junjie Chen
HCST (Peking University), China
{louyiling,chenjunjie}@pku.edu.cn

Lingming Zhang
UT Dallas, USA
lingming.zhang@utdallas.edu

Dan Hao†
Lu Zhang
HCST (Peking University), China
{haodan,zhanglucs}@pku.edu.cn

ABSTRACT
Build systems are essential for modern software development and maintenance since they are widely used to transform source code artifacts into executable software. Previous work shows that build systems break frequently during software evolution. Therefore, automated build-fixing techniques are in huge demand. In this paper we target a mainstream build system, Gradle, which has become the most widely used build system for Java projects in the open-source community (e.g., GitHub). HireBuild, state-of-the-art build-fixing tool for Gradle, has been recently proposed to fix Gradle build failures via mining the history of prior fixes. Although HireBuild has been shown to be effective for fixing real-world Gradle build failures, it was evaluated on only a limited set of build failures, and largely depends on the quality/availability of historical fix information. To investigate the efficacy and limitations of the history-driven build fix, we first construct a new and large build failure dataset from Top-1000 GitHub projects. Then, we evaluate HireBuild on the extended dataset both quantitatively and qualitatively. Inspired by the findings of the study, we propose a simplistic new technique that generates potential patches via searching from the present project under test and external resources rather than the historical fix information. According to our experimental results, the simplistic approach based on present information successfully fixes 2X more reproducible build failures than the state-of-art HireBuild based on historical fix information. Furthermore, our results also reveal various findings/guidelines for future advanced build failure fixing.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.

KEYWORDS
Automated Program Repair, Build System, Build Failure Fixing

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

ISSSTA '19, July 15–19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07...$15.00
https://doi.org/10.1145/3293882.3330578

1 INTRODUCTION
Build systems (e.g., Gradle [4], Ant [2] and Maven [9]) and their corresponding build scripts have been widely used in modern software development to automate the build process. Such build scripts are also frequently updated during software evolution, to be consistent with the changed source code or environment (e.g., third-party libraries and plug-ins). If an inconsistency/bug occurs, a build script may incur build failures. Build failures occur frequently in both commercial and open-source software systems, and may seriously postpone the other activities in software development. For example, in Google, the build failures for Java and C projects occur at the frequency of 28.5% and 38.4%, respectively [57]; on Travis [10], the most popular continuous integration (CI) service, nearly 29% of all the commits suffer from build failures during CI testing [14].

The widespread build-failure problem has gained increasing attention from software engineering researchers, and various studies/techniques on different types of build failures have been conducted/proposed [12, 23, 26, 45, 55, 68]. For example, Al-Kofahi et al. [12] proposed a fault localization approach for Makefile, which collects dynamic execution trace via the concrete build rules and then computes suspiciousness of each statement in Makefile via a ranking algorithm; Macho et al. [45] recently designed three strategies based on the frequently occurring repair types to fix only dependency-related build failures for Maven projects. Among them, HireBuild [23], state-of-the-art general-purpose build-failure fixing technique proposed in ICSE’18, learns fix patterns from successful fixes in history across projects and generates patches by embodying the learned patches. Taking 135 previous build-failure fixes as the training set, HireBuild has been shown to be able to successfully fix 11 (46%) of 24 studied real-world build failures, indicating a promising future for history-driven build-failure fixing.

Despite its effectiveness, HireBuild was only evaluated on a limited dataset. Furthermore, as a history-driven technique, its effectiveness relies on the quality and availability of the training data. Therefore, it is unclear whether HireBuild’s effectiveness can be generalized to other evaluation datasets. In this paper, to fully understand the efficacy and limitations of the history-driven build fixing technique HireBuild and facilitate future build-fix studies, we first build a new and large dataset of 375 real-world build failures from Top-1000 GitHub projects. To our knowledge, this is the largest evaluation in the literature for general-purpose build-failure fixing.
Among the collected 375 build failures, 102 of them are currently reproducible. We thus re-visit the performance of HireBuild on these 102 reproducible build failures. We also perform qualitative manual inspection on the successful and unsuccessful cases of HireBuild to investigate the strengths and limitations of history-driven build-failure fixing. The quantitative results show that HireBuild is able to fix 9 (9%) out of 102 build failures, confirming that HireBuild can indeed commit successful fixes, but in a much lower rate than reported. Meanwhile, based on our qualitative analysis, the fixed build failures by HireBuild usually fall into some fixed patterns, which actually could be also obtained from present information (i.e., present build code and external resources) rather than historical build fix information; the unfixed build failures are mainly due to the inflexible design of fix patterns and patch generation rules, making some patching results hard to generalize to new datasets.

Inspired by the findings of our study, we propose a lightweight build-failure fixing technique, HoBuFF (History-oblivious Build Failure Fixing), which does not rely on history data but instead simply utilizes the present information of the build code, build log and external build-related resources. HoBuFF includes two phases: (1) fault localization [15–17, 37, 38, 40, 50, 51, 74, 76, 77], and (2) patch generation. In particular, in the first fault-localization phase, HoBuFF analyzes error logs of the given build failures to extract error information, and then localizes the possible buggy locations via inter-procedural data-flow analysis; in the second patch-generation phase, HoBuFF generates patch candidates by defining three fixing operators and searching for the fixing ingredients both inside and outside the project (i.e., internal and external resources). We then conduct an empirical comparison between HireBuild and HoBuFF on the extended dataset and find that among the 102 reproducible failures, HoBuFF successfully fixes 18 bugs within less time, including the 8 bugs fixed by HireBuild (HireBuild fixes 9 in total). We also observe that for the build failures that cannot be fixed, HoBuFF can terminate its execution in minutes, whereas HireBuild may take hours. The paper makes the following contributions:

- **Dataset**: A dataset including 375 real-world build failures within 102 reproducible build failures, which is much larger than the state-of-art build-failure datasets and can serve as the benchmark dataset for future build-fix studies.
- **Study**: An extensive study of state-of-the-art history-driven build-failure fixing (HireBuild) on the extended dataset, with detailed manual inspection for both its strengths and limitations.
- **Technique**: A novel build-failure fixing technique (HoBuFF) with only present information (i.e., no requirement for historical data), which utilizes lightweight data-flow analysis and queries internal/external resources to perform build fixing.
- **Implications**: An empirical evaluation of HoBuFF and state-of-the-art HireBuild, which demonstrates that present project information can greatly complement historical build fix information for automated build failure fixing, and also reveals various findings/guidelines for future advanced build-failure fixing.

## 2 BACKGROUND

### 2.1 Build Failure Fixing: Challenges

In this section, we present a build failure in MockIt to/db8a3f3 and its manual fixing patch [5] to illustrate the challenges in fixing build failures for Gradle. Table 1 presents the build failure example with an error information related segment (i.e., error message) from its build log and its manual fixing patch. During the execution of task releaseNeeded, the build process terminates at Line 104, because Line 104 tries to access the property ext.needed, which is not defined on the else branch. To fix this failure, developers add an extra statement ext.needed = false (in green) after Line 102.

Given the build log containing failure related information, the first challenge lies in fault localization in the Gradle script. As the error message in Table 1 indicates, Line 104 is the code location where the build failure is triggered and the build process stops. However, according to the manual patch, the fix is added after Line 102. Line 104 reveals the build failure, but it may not be the root cause for the build failure. Therefore, without identifying all potential root-cause statements, we may not fix a build failure.

Even if the set of root cause statements are identified, there is another challenge: how to generate a correct patch. In particular, to fix this example build failure, an automated fixing technique needs to find out the correct value for the error property ext.needed, which requires the understanding of the program. In other cases, when a build failure is caused by using incorrect values of external resources (e.g., a third-library dependency), an automated fixing technique also requires open knowledge of external resources.

### 2.2 State-of-the-Art HireBuild

As the state-of-art build-failure fixing technique, HireBuild learns fixing patterns from historical build-failure fixes (i.e., pattern extraction) and generates specific patches by predefined rules and filling in concrete values in the pattern (i.e., patch generation).

In the phase of pattern extraction, HireBuild first requires a training set, which is composed of build fixes collected from the history, and then selects several build fixes from the training set whose error messages are similar with the error message of the given build failure. These selected build fixes are regarded as seed fixes. From these seed fixes, HireBuild extracts patterns and then ranks them with some heuristic strategies. For example, for a given fix which changes statement version = 1.4.0 to version = 1.7.0, HireBuild learns a pattern, which is to "replace the constant value (i.e., 1.4.0) in expression version = 1.4.0 with a new value". Note that this new value will be decided later (in the patch generation phase). Moreover, since HireBuild defines patterns by using only
two-level AST expressions (i.e., current and its parent node expressions), its generated fixes often cover only a small span of script code, e.g., often a variable or a continuous segment of script code.

In the phase of patch generation, HireBuild defines several rules for four types of build elements and fills in concrete values into the abstract part of the patterns for these build elements. The four types of elements are (1) identifiers (including task names, block names, variable names), (2) names of Gradle plug-ins and third-party tools/libraries, (3) file paths within the project, and (4) version numbers. Then HireBuild ranks the generated patches based on the similarity between patches and seed fixes, applies them to the code locations matching the patterns, and at last validates them in order.

However, besides the four types of elements considered in HireBuild, there still exist other elements (e.g., project-specific variables), whose concrete values are required in patch generation but are not considered by HireBuild. For all the example in Table 1, `ext._needed` (whose value is true or false) does not belong to any of the four types. For these elements, HireBuild does not define specific rules for concrete value generation, and thus cannot generate candidate patches for the corresponding build failures.

3 STUDY ON HIREEBUILD

As a learning-based technique, HireBuild was evaluated on a very small dataset, including the training and testing data, which may incur overfitting. To alleviate this concern, it is important to re-evaluate its performance on a different and larger dataset. Also, it is essential to manually inspect the cases that HireBuild produces correct fixes or not, to learn its efficacy and possible limitations.

3.1 An Extended Dataset

Why a new dataset is needed? The study of HireBuild [23] uses 135 previous build-failure fixes as training data and a set of 24 reproducible build failures as its evaluation dataset. Such a small data set, especially the evaluation dataset, may bring obvious external threat to validity, and thus the conclusions may not be generalized.

To reduce this threat, we conduct a more extensive study on HireBuild by extending the existing dataset [23]. In particular, we collect extra 375 build failures, among which 102 build failures are reproducible, significantly more than those of the prior dataset.

In this paper, we evaluate build-failure fixing techniques on the dataset of these 102 build failures. This new dataset is abbreviated as the extended dataset hereinafter.

How is the new dataset collected? First, we collect the top-1000 popular Java projects in GitHub [3], and keep only the 411 projects which have been integrated tested in Travis system according to whether the `.travis.yml` file is in the project.

Second, we collect build bugs based on the history data of these projects. For each of these 411 projects, we first identify a commit (denoted as $V_F$) whose build status is failed and whose immediately successive commit (denoted as $V_P$) is passed. As the failure of $V_F$ may come from either the build script or the source code, we keep only the commit $V_F$ whose changes from $V_F$ to $V_P$ occur on Gradle files alone. Among these 501 resulting commits (which are actually commits containing build bugs), we manually remove those whose modifications on Gradle files do not influence the build results (e.g., documentation modifications or semantic-equivalent modifications on Gradle files) and then in total have 403 build failures, each of which is suited with a failed commit and a successive passed commit.

To avoid overlap between this newly constructed dataset and the prior dataset [23], we further remove build failures already in the prior dataset, and finally have a new dataset of 375 build failures.

Among the 375 build bugs, we successfully reproduce 102 build failures. A build bug is reproducible when its failed commit still fails (due to the same reason) and its originally-passed commit still passes. We find it more challenging to reproduce build failures than general program bugs since build failures associate tightly with external resources (which differ among different time stamps), and are also susceptible to internal resources (which differ among machines/environments): (1) The dependent libraries which were originally missing in repository, now are added to repository, or the dependent libraries which originally contained flaws, now are fixed. In this case, the originally-failed commits no longer fail anymore. (2) The external resource changes of the libraries which were not relevant to the build failure in the past, now also cause the build to fail (but due to a different reason). In this case, the originally-passed commits no longer pass now. (3) The build failures are caused by internal machine resources (e.g., build process crash due to specific memory/process status), and are hard to reproduce in a new machine. (4) The build failures are simply flaky (e.g., due to flaky tests [44]), and are hard to reproduce.

Table 2 presents the basic information of the extended dataset, which shows the scale of the extended data set compared with that of the prior evaluation dataset. Noted that, there is no overlap between the build failures in the extended dataset and the prior one [23] (since we intentionally removed such overlapped failures).

3.2 Research Questions
We investigate following research questions for studying HireBuild:

- **RQ1**: How does HireBuild perform on the extended dataset in terms of the number of fixed build failures?
- **RQ2**: Why does HireBuild succeed to fix some build failures?
- **RQ3**: Why does HireBuild fail to fix some build failures?

For an unbiased study of Hirebuild, we ask for the original implementation of HireBuild from the authors and directly use it for our study. Moreover, we take the setting of Hirebuild used in the previous work [23], i.e., using the same training dataset (135 build failures from its dataset) and the setting parameters (e.g., the number of seed build fixes is 5).

3.3 Results and Analysis

3.3.1 RQ1: Number of Fixed Failures by HireBuild. Among 102 reproducible build failures in the extended dataset, HireBuild fixes only 9 of them (9%), which is much lower than the fixing rate reported on the prior dataset (i.e., 46% [23]). In other words, HireBuild does not perform as well as it appears in its original dataset, and may suffer from the overfitting problem. Besides the quantitative analysis, it is also interesting to investigate the performance of HireBuild in details, including its successfully-fixed/unfixed cases.
Table 3: An Example Successful Patch by HireBuild

<table>
<thead>
<tr>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>What went wrong:</td>
</tr>
<tr>
<td>&gt; A problem occurred evaluating root project 'AnimeTaste'</td>
</tr>
<tr>
<td>&gt; Couldn't resolve all dependencies for config 'debugCompile'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Patch Generated by HireBuild</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 compile com.android.support:appcompat-v7:22.0.0</td>
</tr>
<tr>
<td>31 compile com.github.johnpapa:supertoasts:1.3.4</td>
</tr>
<tr>
<td>32 compile com.afollestad.material-dialogs:0.8.6.2</td>
</tr>
</tbody>
</table>

3.3.2 RQ2: Successfully-fixed Cases. For the 9 fixed build failures, most of them are fixed by using a rigid pattern in a straightforward way, and history information is not very necessary. For example, Table 3 shows a build failure caused by an unresolved third-library. The error message suggests that the third-library named com.afollestad.material-dialogs with version 0.6.3.1 could not be resolved. To fix this failure, HireBuild learns a pattern based on similar build-failure fixes related to library resolving, which is to update the constant value in the expression starting with keyword compile. However, as also shown by this table, compile is a very common keyword in Gradle scripts so that many expression starts with compile. To further localize the faulty code, HireBuild designs extra ranking rules which assign high priority to the expression sharing similar tokens in error message (i.e., com.afollestad.material-dialogs). To generate patches then, after recognizing the element as a third-library with predefined rules, HireBuild searches in the Gradle central repository to find proper version values for the library.

The fixing process of this example suggests that the patterns learned by HireBuild from historical data (i.e., updating the constant value in the expression starting with keyword compile) actually play a marginal role to the final success of fixing. On the contrary, the present information, including the error message, script code itself, and the external resources (e.g., the Gradle central repository), may be sufficient for fixing such failures.

3.3.3 RQ3: Unfixed Cases. HireBuild fails to fix 93 failures, because its inflexible pattern generation and application mechanism hampers (1) localizing the faulty code, (2) generating correct patches.

Unsuccessful Fault Localization. As introduced in Section 2.2, HireBuild can not distinguish potential faulty code in the process of fault localization. In particular, HireBuild regards all the expressions matching the patterns as faulty code (i.e., the place where to apply the patterns then). In detail, if the learned pattern is "to update the constant value of an expression which starts with version =", only the expressions starting with "version =" are considered faulty code. This strategy works well only when the faulty code exactly matches the learned patterns.

However, the faulty code does not always match the learned patterns in such a rigid way. For the illustration example in Section 2, to fix this build failure, HireBuild needs to learn a pattern "inserting an expression: ext..needed = false". However, the essential variable ext..needed is named in a project-specific way, and HireBuild can hardly learn such a pattern from the training dataset.

To sum up, HireBuild generates inflexible patch patterns, which can hardly deal with project-specific failures.

Unsuccessful Patch Generation. To generate patches, HireBuild embodies rules for only four type of elements in specific patch generation. However, besides these four types of elements, Gradle scripts may contain other elements, e.g., user-defined variables whose values are Boolean or Strings, which do not belong to any of the four types. For example in Table 1, ext..needed (whose value is true or false) does not belong to any of the four types. For these elements, HireBuild does not design specific rules for concrete value generation, and thus cannot generate candidate patches for the corresponding build failures.

To sum up, HireBuild only embodies patch generation rules for specific elements, and thus cannot generate patches for all build failures, even if HireBuild precisely localizes the buggy code.

3.4 Enlightenment

According to the findings of above research questions, we could infer some guidelines for automatically fixing build failures:

- **Necessity of using historical data.** Historical fixing data are not the indispensable factor for build-failure fixing techniques. On the contrary, it is actually more essential to make good use of the present information (i.e., present script code, present build log and internal/external resources).

- **Feasibility of using present data.** Present script code, build log are often available for a given build failure. Besides, lots of program analysis techniques could be adapted to analyze the script code. Furthermore, build logs are very well-structured and thus allow plain error information extraction. As for resources, there are official documents and repositories stored in a well-structured for automated reference.

- **Analyzing more patterns for build code.** The patterns extracted by HireBuild are inflexible because it keeps little program information. It implies that, analyzing more fix patterns from build code could help draw pivotal clues for build-failure fixing.

- **Considering more elements in build code.** The limited types of elements in build code considered for patch generation also cause the limitation of HireBuild’s efficacy, which implies that, a more general/systematic build-failure fixing approach also requires considering more script elements.

4 A NEW TECHNIQUE: HOBUFF

Inspired by the findings of study on HireBuild, we propose a lightweight build-failure fixing technique, named HoBuFF (History-oblivious Build Failure Fixing), which does not take historical fixes as input, but utilizes present information in a more exhaustive way.

Build-failure Fixing Problem Definition. At a high level, a Gradle build script can be regarded as a collection of configurations, each of which consists of a configuration element and its value. Most build failures can be attributed to incorrect configurations, e.g., assigning a wrong value to a configuration element or missing a configuration.

More formally, any build script can be denoted as a set \(C = \{c_1, c_2, \ldots, c_n\}\), where \(C\) denotes a build script and \(c_i = <e_i, \psi_i >\) (1 ≤ i ≤ n) is a configuration implicitly or explicitly claimed in \(C\). Here, \(e_i\) is a configuration element and \(\psi_i\) is the value of the element. Supposed that a build failure occurs when using \(C\), the problem of build failure fixing is to generate a new build script
When a build failure occurs, a build log records the corresponding error information via lightweight inter-procedural data-flow analysis. Figure 1 shows the overview of HoBuFF. In the first phase of fault localization, HoBuFF extracts error information by analyzing build logs, and then localizes the potential buggy code by applying lightweight data-flow analysis (Section 4.1). In the second phase of patch generation, HoBuFF designs fixing operators and searches for the fixing ingredients, so as to generate patch candidates (Section 4.2). For ease of understanding, we use the example presented in Section 2 to illustrate our approach throughout this section.

### 4.1 Dataflow-Based Fault Localization

When a build failure occurs, a build log records the corresponding error information, which is helpful to manually localize the root cause of the build failure. Therefore, the first step of HoBuFF for fault localization is to extract the error information from the build log. Based on the extracted error information, HoBuFF then localizes the bug-revealing statement(s), which is the statement(s) in the build script that exposes the build failure during the build process. Finally, HoBuFF traces the root cause from the bug-revealing statement(s) via lightweight inter-procedural data-flow analysis. Figure 2 shows the workflow of the fault-localization process on our example.

#### 4.1.1 Error Information Extraction

A build log tends to record much information involving various stages of a build process, such as initialization and task execution. Therefore, HoBuFF first parses the build log to extract the message related to the build failure, called error message. Due to the standard form of Gradle build logs, there exist error-indicating headers in the log to mark the error message. As shown in the error message of the illustration example, there are two error-indicating headers: (1) “What went wrong:” explaining the symptom of the build failure; (2) “Where:” indicating the location of the bug-revealing statement(s). Following the existing work [23], HoBuFF utilizes the error-indicating headers to extract the error message from a build log. In the error message, Gradle always tries to report in which project and in which task, the build failure occurs, which are reported in standard form and

#### 4.1.2 Bug-Revealing Statement Identification

The bug-revealing statement(s) is responsible to expose the build failure, and may not be the root cause of the failure, but it is actually very important to help identify the root cause. The bug-revealing statement(s) can be identified based on the aforementioned extracted error information. If the last type of information, i.e., location of the bug-revealing statement(s), exists, HoBuFF is able to directly find the bug-revealing statement in the build script according to the line number. As shown in the illustration example, the bug-revealing statement is identified at Line 104. Otherwise, HoBuFF uses the other types of information to infer the bug-revealing statement in the build script.
and the name of the extracted configuration element, and then identifies the statement(s) containing the variable with the smallest distance as the bug-revealing statement(s). If there are more than one statement satisfying the preceding condition, HoBuFF identifies the ones within the extracted project or task.

4.1.3 Root Cause Localization. To identify the root cause in the build script, HoBuFF performs inter-procedural data-flow analysis to trace backward from a bug-revealing statement(s). Note that we only consider data-flow dependencies (ignoring control-flow dependencies) to avoid over-approximations \cite{59, 70} for lightweight analysis. First, HoBuFF constructs an inter-procedural control-flow graph, and annotates it with the data dependency information computed via reaching definition analysis \cite{49}. Since Gradle is a Groovy-based domain-specific language and it defines a set of its own rules to serve as a build tool, its analysis is slightly different from widely-used programming languages such as C++ and Java. Therefore, we perform inter-procedural, context-insensitive, and field-sensitive dataflow analysis considering the following specific features of Gradle files: (1) variable definitions in one Gradle script file (or tasks) often use variables defined in other Gradle files (or tasks), thus we perform inter-procedural analysis; (2) there are not intensive function invocations in Gradle files (e.g., Gradle scripts largely reply on task dependencies/sequences rather than method invocations to implement the build logic), thus we perform context-insensitive analysis; (3) fields in aggregate structure variables are widely used in Gradle files (e.g., ext. needed in Table 1), thus we perform field-sensitive analysis. Note that if a variable V is used without definition for some program path, an implicit statement \( V = \text{null} \) would be inserted into the location which is not reached by any definition of V and is also the closest to the statement using V. Then there should be an edge between the inserted statement and the statement using V. For example, there is missing definition in else branch for Line 104 in Table 1, thus we insert statement \( \text{ext. needed} = \text{null} \) in Line 102, and add an edge from Line 102 to Line 104. The constructed data dependencies for the example in Table 1 is shown in Figure 2. Then, based on the graph, HoBuFF identifies all the statements affecting the bug-revealing statement (including the bug-revealing statement) as potential root-cause statements, which is the output of the fault-localization process. For example, Lines 98, 100, 102, and 104 are identified as the potential root-cause statements of the build failure.

4.2 Search-Based Patch Generation

To fix a build failure, HoBuFF uses a search-based approach to generate patch candidates for each root-cause statement, and then applies these patches one by one to the buggy Gradle script. If the build failure disappears when applying some patch, HoBuFF regards this patch as valid. The fixing process is conducted continuously until all patch candidates have been applied or a valid patch is found. In the following, we first introduce the components required by search-based patch generation, i.e., fixing operators (Section 4.2.1) and fixing ingredients (Section 4.2.2). Then, we present the overall process of patch candidates generation (Section 4.2.3).

4.2.1 Fixing Operators. Given a buggy or missing configuration \( c_b = \langle e_b, \psi_b \rangle \), we define three fixing operators in HoBuFF following existing work for source-code repair \cite{29, 42, 67}:

- **Update**: Update \( c_b \) by replacing \( \psi_b \) with the correct value \( \psi_b^+ \), where \( \psi_b^+ \) is the ingredient. Note that how to define ingredients will be introduced in the following subsection.
- **Insertion**: Insert a configuration \( c_b \), where \( e_b \) is decided through fault localization, whereas \( \psi_b^+ \) is the ingredient to be introduced.
- **Deletion**: Delete the configuration \( c_b \), where no ingredient is required. In this case, \( c_b = \text{null} \) (\( c_b \) is removed from C). To avoid syntax problem, we further analyze the data dependencies from \( c_b \) to delete all the affected statements as well.

For each root-cause statement, HoBuFF applies these operators in the order of Update-Insertion-Deletion. As the previous study on source-code repair \cite{67} shows, “Update” is the most widely-used operator in manual bug repairs while “Deletion” is the least used fixing operator. In particular, if the root-cause statement is an inserted null expression (e.g., \( \text{ext. need} = \text{null} \) in Figure 2), only “Insertion” can be applied to it since it is an implicit statement.

4.2.2 Fixing Ingredients. Both “Update” and “Insertion” require fixing ingredients. We propose a search-based approach to find the correct ingredients for these operators.

We classify the configuration elements into two types and decide their values (i.e., ingredients) in different ways. The first type of configuration elements is defined within the project, e.g., properties or files, called **internal elements**; whereas the second type of configuration elements is related to external libraries, e.g., third-party tools and dependencies, called **external elements**. For internal elements, HoBuFF searches ingredients inside the project, i.e., internal searching in short. For external elements, HoBuFF searches ingredients outside the project, i.e., external searching in short.

HoBuFF with internal searching, is to search all the values that are assigned to the same configuration element within the whole project. For example, there are two ingredients found in this way for the illustration example: \( \text{ext. needed} = \text{false} \); \( \text{ext. needed} = \text{true} \); HoBuFF with external searching, is to search the values from external resources. Here we consider three kinds of external resources: (i) Gradle central repository \cite{5} recording most of third-party dependencies; (ii) Gradle DSL document \cite{6} recording Gradle types and their corresponding properties and potential values; (iii) Android DSL document \cite{1} recording most of Android-related plugins and their corresponding properties. We also consider Android DSL because Gradle is the official build tool for Android and Gradle build scripts usually have dependencies with Android.

Since the external resources are recorded in a well-structured form, HoBuFF is able to collect the information for these external resources in advance. Due to the closure characteristic of Gradle, for each item in the external resources, it can be represented in a sequence like \( \langle \text{prefix}_1, \text{prefix}_2, ..., \text{prefix}_n : \text{valueType} > \). For example, from the Android DSL document, we could collect item like \( \langle \text{android lintOptions . abortOnError : Boolean} \rangle \), which means in android block and its sub-block lintOptions, there is an element abortOnError, whose value type is Boolean and has two optional values: true or false.

Given a searching keyword, HoBuFF tries to match it within collected sequences, and retrieves the value type and prefixes for
the keyword. For example, if HoBuFF finds the buggy element named `l1nt`, HoBuFF searches for `l1nt` within collected sequences and finds the related one `l1ntOptions` and its value type. Based on the information, HoBuFF could generate several fixing ingredients, such as, `android.lintOptions.abortOnError = true` and `android.lintOptions.abortOnWarning = false`. The transformed Gradle code from these ingredients can refer to Table 6. In this way, a set of ingredients can be collected from external resources.

4.2.3 Patch candidate generation. The input of the patch-candidate-generation process is a set of localized root-cause statements and the output is a list of patch candidates for the build failure. More specifically, the patch-candidate-generation process can be unscrambled as the following three layers: (1) for each root-cause statement, HoBuFF first decides which fixing operators should be applied according to whether the statement is null definition or not. If the statement is null definition, only “Insertion” is considered; otherwise, all three fixing operators are applied in the order of Update-Insertion-Deletion, respectively; (2) for each fixing operator, HoBuFF directly generates a patch candidate if it is “Deletion”, while HoBuFF generates fixing ingredients via internal or external searching if it is “Insertion” or “Update”; (3) for each fixing ingredient, HoBuFF generates a patch and validates it.

5 COMPARISON EVALUATION

5.1 Research Questions

- **RQ4**: How does HoBuFF perform in terms of the number of fixed build failures?
- **RQ5**: How does HoBuFF perform in terms of the build-failure fixing time and the number of candidate patches?

While RQ4 focuses on the effectiveness, RQ5 studies the time HoBuFF costs and the number of candidate patches validated before a correct patch is found. These measurements are widely used in the existing work in program repair [53, 65, 67].

5.2 Implementation, Environment, and Process

To implement HoBuFF, we use Groovy AST APIs [7] to systematically analyze/modify Gradle build scripts. We conduct all experiments on a computer with 64 Intel(R) Xeon e5 CPU Cores, 128GB Memory, and Ubuntu 14.04.1. The tool/dataset can be found in our website: https://sites.google.com/site/hobuff2019.

To investigate the performance of HoBuFF, for each build failure, we first apply HoBuFF to its corresponding buggy build script and collect a list of patches generated by HoBuFF. For each patch, we follow previous work [23] to validate whether it is correct based on the following two criteria: (1) the build task finishes successfully; and (2) the size of compiled files is the same as the size of compiled files generated by the manual patch. Finally, we count the number of build failures that HoBuFF can successfully fix. For each fixed build failure, we also record the time spent by HoBuFF and the number of validated candidate patches. To compared with the state-of-art, we also record the same measurements for HireBuild.

<table>
<thead>
<tr>
<th>Build Failure Category</th>
<th>HoBuFF</th>
<th>HireBuild</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Element Related</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>External Element Related</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>12</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4: Build Failures Fixed by HoBuFF/HireBuild

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Patch Generated by HoBuFF</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt; A problem occurred evaluating project 'app'</code></td>
<td><code>android.android.lintOptions.abortOnError = true</code></td>
</tr>
<tr>
<td><code>&gt; /home/travis/build/yydcdut/PhotoNoter/app/release.properties (No such file or directory)</code></td>
<td><code>android.android.lintOptions.abortOnError = true</code></td>
</tr>
</tbody>
</table>

Table 5: Build Failure with Non-existing File

5.3 RQ4: Build-Failure Fixing Effectiveness

Among the 102 reproducible build failures, HoBuFF successfully fixes 18 of them (18%), whereas the state-of-art HireBuild fixes only 9 of them (9%), indicating the superiority of simply using the present project information rather than using the historical fix information for build-failure fixing. In addition, among the 9 build failures fixed by HireBuild, 8 are also fixed by HoBuFF, which implies that the new simplistic approach is able to fix most of the build failures HireBuild fixes. To investigate the contribution of each component of HoBuFF, we further combine fault localization of HoBuFF with patch generation of HireBuild, fault localization of HireBuild with patch generation of HoBuFF on the 18 failures fixed by HoBuFF. The results show that the former only fixes 12 failures while the latter only fixes 8 failures, demonstrating the contribution for each component of HoBuFF. More qualitative analysis on the capability of HoBuFF over HireBuild can be found as follows.

5.3.1 Case Analysis for Successfully-Fixed Failures. To facilitate analysis, we categorize build failures successfully fixed by HoBuFF according to the configuration locations that the corresponding fixes deal with. Table 4 presents the results of each category, where Columns 2 and 3 show the number of build failures fixed by HoBuFF and HireBuild within each failure category, Column 4 presents the number of build failures fixed by both of these two techniques. **Internal-element-related failures** refer to the build failures resulting from wrong values of internal configuration elements. The values of internal configuration elements are often specific to the project so that their values usually vary among projects. Properties and files are common internal elements. Table 5 presents such a real build failure in our study and its fix generated by HoBuFF. From the error message, such a failure is caused by non-existing local files. HoBuFF identifies the buggy configuration in Line 56 as the bug-inducing statement based on the file path. As HoBuFF cannot find fixing ingredients for insertion and update operators on this bug-inducing statement, it generates a patch by applying deletion operators to Line 56 and its affected code (Lines 57-60). Note that this patch is the same as the corresponding manual patch.

According to Table 4, HireBuild does not fix any internal element-related failures. HireBuild learns patches across projects and different projects define different internal configurations (different element names and different element values), HireBuild cannot learn how to fix such category of failures from the history data. In contrast, HoBuFF is good at fixing these failures, since HoBuFF performs data-flow analysis for precise localization and internal searching for valid patch generation. **External-element-related failures** refer to the failures resulting from improper values of elements in external configurations, such
As the options of third-party plug-ins. Among these 10 build failures successfully fixed by HoBuFF, we further found that Gradle repository, Gradle DSL document, and Android DSL document contribute to 20%, 20% and 60% of the cases, respectively. Table 6 presents such a real-world build failure in our study and its fix generated by HoBuFF. This failure is caused by error option of lint component. HoBuFF identifies the buggy configuration in Line 94 as the bug-revealing statement and the root cause. Then, HoBuFF applies external searching and find lintOptions owns a set of options (e.g., abortOnError, absolutePaths). Lastly, HoBuFF enumerates all values of these options since they are Boolean type, and combines them with the insertion operator to generate candidate patches. Note that although disabling lint option seems tricky, we observed that the developer(s) also did exactly the same “lazy” fix.

Table 7 presents another real-world build failure in our study and its fixes generated by both HoBuFF and HireBuild. In particular, the error message shows that the value of “Build Tools revision” is wrong. With this message, HoBuFF finds the buggy configuration in the build file, which is Line 25 in the table, i.e., buildToolsVersion = _buildToolsVersion. buildToolsVersion is an option of Android plug-in, and it declares the version number of build tools. However, this line is not the root cause and only triggers the failure. Based on the fault localization component, HoBuFF identifies the real root cause of this build failure, which is Line 3 (i.e., buildToolsVersion = _buildToolsVersion). For comparison, we also list the patch generated by HireBuild in the last row, which is different from the manual patch. Although this patch can also fix this build failure (and has been counted as a successful fix), HireBuild brings dead code (i.e., Line 3: buildToolsVersion = 26.0.2 becomes dead code) to the build file and thus brings bad code smell. Moreover, on the other hand, this case also demonstrates the importance of precise fault localization in build-failure fixing.

5.3.2 Case Analysis for the Remaining Failures. Although HoBuFF outperforms the state-of-art HireBuild to a large extent, it does not fix all the build failures. Therefore, in this subsection we investigate the remaining build failures that cannot be fixed by our approach to learn its limitation in fault localization and patch generation.

Among the 84 build failures that cannot be fixed by HoBuFF, 58 cannot be fixed due to the early fault localization phase of HoBuFF, whereas 26 cannot be fixed due to the latter patch generation phase.

Fault localization limitation. For most unsuccessful cases caused by the limitation of fault localization, we further categorize them according to the unsuccessful reasons and corresponding build failure types, shown in Table 8.

We observe that most of the unsuccessful cases (56 out of 58) get stuck in error information extraction. For 43 cases, error messages are too abstract to contain detailed information for localization, so that HoBuFF can not extract buggy configuration information at all. This kind of error messages often comes from the build failures with test failures, compilation errors, abnormal processes, or startup failures. We list an example with error message and related manual fix in Table 9. From this example, even experienced developers may not be able to find the relation between the error message and the buggy code. Since HoBuFF can not extract buggy configuration from the error message, one potential solution is to utilize extra information, such as stack-trace or report files, which can be extended to HoBuFF in the future. Furthermore, HireBuild also fails to fix these failures, since such fixes can hardly be learned from the history either. For 6 cases of indirect error messages due to runtime exceptions and publish failures, their error messages contain scrap and indirect information composing in a complex way which is hard to extract error information. For 7 cases of unrelated error messages due to file/dependency missing, the configuration element reported by its error message is not the real cause of the build failure. Table 10 shows an example. With the error message, HoBuFF identifies Line 7 as the buggy statement and tries to assign a correct value for this dependency. However, the real cause is that the build script does not include a proper central repository (i.e., jcenter()) and mavenCentral doesn’t contain the needed com.android.tools.build.gradle above 2.3.1. HoBuFF considers jcenter as the default repository for searching third-party dependency without considering the buggy build script missing including jcenter. To deal with such a build failure, HoBuFF needs to include some extra common sense rules during the process of root cause localization. Note that this is the only build failure in our study that is fixed by HireBuild but not by HoBuFF. HireBuild can fix this failure because its training set has a very similar failure which is fixed in this way, which implies that history and present information are complementary and we will further improve HoBuFF by considering history information in the future.

For the left 2 cases, HoBuFF cannot map the buggy configuration in the build script because the bug-revealing statement lies in a remote source file which is not accessible (i.e., not in local path).
failures because their fixes require to modify plenty of lines in the build file and the modification involves complex logic, rather than reset the value of configuration elements. Table 11 shows one example case. In particular, HoBuFF extracts suspicious configuration `<pom_name, null>` and tries to fix the value of the property `pom_name` by update, insertion and deletion operators but the correct patch is not generated at all. The last row shows the manual patch for this build failure, which actually includes a large amount of complex modification (i.e., adding 59 lines and deleting 58 lines). Moreover, such modification requires comprehensive understanding of the project, which is beyond the current capability of HoBuFF. Such a failure whose fixing requires many complex modifications is still an open problem in both source-code repair [5, 13, 22, 30–33, 39, 41, 48, 58, 64, 71–73] and build-failure repair [23, 45], which may be further explored in the future.

### 5.4 RQ5: Build-Failure Fixing Efficiency

Besides the number of fixed bugs, it is also interesting to study the time spent by a bug-fixing technique. Even if the existing bug-fixing techniques (including HireBuild and HoBuFF) can fix the same build failures, a technique with quick response (i.e., fixing a build failure quickly and notifying incapability quickly) is preferable. Therefore, in this subsection we analyze bug-fixing time by considering both successfully-fixed failures and unfixed failures. For the failures successfully fixed, HoBuFF spends 156 seconds on average, indicating the efficiency of HoBuFF in fixing real-world build failures. We further draw a violin plot to compare the fixing time between HoBuFF and HireBuild in Figure 3. From this figure, neither technique spends long time in successfully-fixed cases (i.e., less than 800 seconds). Moreover, although their fixing time is distributed in a close range, the fixing time of HoBuFF concentrates in a smaller range (i.e., less than 200 seconds) than HireBuild, demonstrating the superiority of HoBuFF in terms of the fixing time for successful fixes. For the 8 failures fixed both by HoBuFF and HireBuild, HoBuFF takes 118 seconds on average and HireBuild takes longer, 196 seconds. For the remaining unfixed failures, HoBuFF consumes 606 seconds on average and 1110 seconds at most, before notifying its incapability. However, HireBuild requires much longer time (i.e., 5 hours on average and 14 hours at most) to report its incapability. The reason is that HireBuild’s learning process creates many unnecessary patches (shown in the next paragraph).

We also analyzed the Number of Candidate Patches (abbreviated as NCP) generated and evaluated before a valid patch is found in the successful repair cases. Similar to the existing work on program repair [53], a smaller NCP score indicates less patches are validated during the repair process, which implies high performance. Figure 4 presents a violin plot on the distribution of NCP scores for HoBuFF and HireBuild. From this figure, both techniques distribute in a small range. The NCP scores of HoBuFF are distributed in a smaller range (i.e., 1-5). Such conclusions are consistent with the observations in time consumption. That is, as the number of candidate patches validated before finding the correct patch is small, HoBuFF can fix the build failure quickly. Besides, we also investigate the generated patches when the corresponding repair technique does not fix a build failure. Since none of these patches are correct, it is also preferable to generate a small number of candidate patches. On average, HoBuFF and HireBuild generate 77 and 270 patches, respectively, further demonstrating the efficiency of HoBuFF.

### 5.5 Threats to Validity

The threat to internal validity lies in the implementation of the build failure fixing techniques studied in the experimental study. To reduce this threat, we reused the code of HireBuild and used the mature Groovy Parsing APIs to implement HoBuFF. Moreover, the first two authors manually reviewed HoBuFF code carefully.

The threat to external validity mainly lies in the datasets used. Before this study, Hassan et al. [23] have released a dataset of 175 Gradle build bugs when proposing the state-of-art HireBuild. In their work, 135 bugs have been already used as the training set for HireBuild, and among the left 40 bugs, they used the 24 successfully-reproduced bugs for evaluation. We tried to reproduce the 24 bugs as the process described in Section 3.1 and only successfully reproduced 5 bugs due to various reasons mentioned in Section 3.1. We have applied HoBuFF and HireBuild on the 5 reproducible bugs and they both can successfully fix 3 of them. We also build a new dataset, which is the largest dataset of reproducible real build failures from GitHub, to ensure that our experimental results can more
likely generalize to more build failures in the wild compared to prior work. Furthermore, our large dataset can be viewed as two different sub-datasets each with half of the bugs based on the build timestamp. We observe that the earlier half already includes all the necessary information to design HoBuFF. We then can view the later half as the test set. In this way, HireBuild fixes 7/2 bugs, while HoBuFF fixes 10/8 bugs for the earlier/later subset, further demonstrating the generality of HoBuFF over prior work.

The threat to construct validity mainly lies in the metrics used. To reduce the threat, we adopt the widely used metrics in program-repair literature, e.g., the number/ratio of fixed bugs and time cost [67, 69]. Note that following prior work [23], we did not compare generated and manual patches for checking patch correctness since there can be over one solution to fix a build failure (Section 5.2). To further reduce this threat, we manually checked all 18 HoBuFF patches: 10/3 patches are syntactically/semantically equivalent to manual patches, and the rest 5 are all valid alternative patches.

6 RELATED WORK

Program Repair. Automatic program repair is now attracting increasing research interests. There exist various techniques for fixing general bugs [13, 21, 24, 30–33, 48, 58, 64, 71, 72], concurrent bugs [39, 41, 73], and even tests [20, 60]. This section mainly discusses the closely related Generate&Validate (G&V) techniques.

Search-based techniques explore the space of fix templates and validate them heuristically. GenProg [35], one of the earliest and representative search-based APR techniques, searches for correct patches via genetic programming. To reduce the repair time cost of GenProg, RSRepair [52] searches among all candidates randomly, while AE [66] uses a deterministic repair algorithm. To generate high-quality fix patterns, PAR [29] learns various types of fixing templates via manually reviewing human written patches, and leverages them during candidate patch generation. Recently, more and more search-based techniques have been proposed: HDRepair [34] automatically mines historical data to help search correct patches; Elixir [56] uses machine learning to prioritize patches for faster repair; CapGen [67] utilizes context information to rank mutation operators [27, 75] and patches for fast patch generation; SketchFix [25] and JAID [18] reduce patch generation costs via sketching and meta-program encoding, respectively; SimFix [28] searches for similar code snippets from the current project under test for potential fixes; PraPR [21] recently demonstrates that even simple template-based APR mutators can outperform state-of-the-art APR techniques, and shows that bytecode-level repair can achieve over 10X speedup over existing techniques.

Semantics-based APR techniques use constraints to generate correct-by-construction patches via formal verification or specifications. SPR [42] leverages mutation operators to generate candidate patches and also applies condition synthesis via symbolic execution. Prophet [43] automatically learns from correct patches to rank candidate patches generated by SPR for faster repair. SemFix [48] derives repair constraints from tests and solves the repair constraints to generate valid patches. Angleix [47] is a more recent lightweight semantics-based repair technique that scales up to large programs.

HoBuFF can also be categorized as search-based techniques, but it is different from existing techniques: (1) HoBuFF targets at build code fixing, while the existing techniques target at source code; (2) HoBuFF employs both internal and external searching during patch generation, and also constructs external knowledge graph from official documents/sources for candidate generation; (3) HoBuFF utilizes lightweight NLP techniques and dataflow analysis to reduce the search space for the specific build-failure fixing problem.

Build System Maintenance. Recently, the research work on build system maintenance mainly (but not limited to) focuses on empirical study of build-failures and build-failure detection/debugging. In particular, Sulir et al. [61] and Tufano et al. [63] investigated build errors from open source projects in Java. Hyunmin et al. [57] investigated build errors at Google. Both studies demonstrated that build failures occur frequently in practice. To facilitate build failure detection, Wolf et al. [19] proposed to predict build failures via social network analysis on developer communication. Tamrawi et al. [62] proposed a build code smell detection approach, which statically analyzes build code via symbolic evaluation. Besides, Adams et al. [11] utilized a flexible directed acyclic graph to model dependency graph for a build system, which may ease the understanding of a build system so as to reduce the possibility of build failure occurrence. Few work in the literature studies how to automatically fix a build failure. Al-Kofahi et al. [12] proposed a fault localization approach for Makefile, which collects and analyzes dynamic execution trace of build code for precise fault localization. Macho et al. [45] designed three strategies based on frequently occurring repair types to fix only dependency-related build failures for Maven projects. Recently, Hassan and Wang [23] proposed a general-purpose history-based automatic build-failure fixing approach, HireBuild, which learns fixing patterns from historical fixes and feeds them into the buggy script. Our HoBuFF also targets at general-purpose build-failure fixing. However, HireBuild focuses on learning from the history, while HoBuFF searches from the present projects/resources.

7 CONCLUSION

In this paper, we attempt to investigate the potential strengths and limitations of state-of-the-art history-driven build-failure fixing technique, HireBuild. To this end, we construct a new and large real-world build-failure dataset from Top-1000 GitHub projects. Then, we evaluate HireBuild on the extended dataset with both quantitative and qualitative analysis. Inspired by the findings of the study, we propose a history-oblivious technique, HoBuFF, which locates buggy configurations through lightweight dataflow analysis and then generates potential patches via searching from the present project under test and external resources (rather than the historical fix information). The experimental results demonstrate that the simplistic approach based on present information successfully fixes 2X more reproducible build failures than the state-of-art HireBuild and is much faster. Furthermore, our results also reveal various findings/guidelines for future advanced build failure fixing.

ACKNOWLEDGEMENTS

This work was partially supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, the National Natural Science Foundation of China under Grant Nos. 61872008 and 61861130363, and the National Science Foundation under Grant Nos. CCF-1566589 and CCF-1763906.
REFERENCES


[18] Timo Wolf Adrian Schröter Daniela Damian and Thanh Nguyen. [n. d.]. Predicting Build Failures using Social Network Analysis on Developer Communication. [in d.].


