HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts

Foyzul Hassan
The University of Texas at San Antonio
foyzul.hassan@my.utsa.edu

Xiaoyin Wang
The University of Texas at San Antonio
xiaoyin.wang@utsa.edu

ABSTRACT

Advancements in software build tools such as Maven reduce build management effort, but developers still need specialized knowledge and long time to maintain build scripts and resolve build failures. More recent build tools such as Gradle give developers greater extent of customization flexibility, but can be even more difficult to maintain. According to the TravisTorrent dataset of open-source software continuous integration, 22% of code commits include changes in build script files to maintain build scripts or to resolve build failures. Automated program repair techniques have great potential to reduce cost of resolving software failures, but the existing techniques mostly focus on repairing source code so that they cannot directly help resolving software build failures. To address this limitation, we propose HireBuild: History-Driven Repair of Build Scripts, the first approach to automatic patch generation for build scripts, using fix patterns automatically generated from existing build script fixes and recommending fix patterns based on build log similarity. From TravisTorrent dataset, we extracted 175 build failures and their corresponding fixes which revise Gradle build scripts. Among these 175 build failures, we used the 135 earlier build fixes for automatic fix-pattern generation and the more recent 40 build failures (fixes) for evaluation of our approach. Our experiment shows that our approach can fix 11 of 24 reproducible build failures, or 45% of the reproducible build failures, within comparable time of manual fixes.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; *Maintaining software*;

KEYWORDS

Patch Generation, Software Build Scripts, Build Logs

ACM Reference Format:
Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180181

1 INTRODUCTION

Most well maintained software projects use build tools, such as Ant [44], Maven [29] and Gradle [18] to automate the software building and testing process. Using these tools, developers can describe the build process of their projects with build scripts such as `build.xml` for Ant, `pom.xml` for Maven, and `build.gradle` for Gradle. With growing software size and functionality, build scripts can be complicated [25] and may need frequent maintenance [16]. As software evolves, developers make changes to their code, test cases, system configuration, and dependencies, which may all lead to necessary changes in the build script. Adams et al. [4] found strong co-evolutionary relationship between source code and build script in their study. Since build scripts need to be synchronized with source code and the whole build environment, neglecting such changes in build scripts often leads to build failures.

According to our statistics on TravisTorrent [9] dataset on the continuous integration of open-source software projects, 29% of code commits fail to go through a successful build on the integration server. Seo *et al.* [35] also mentioned a similar build failure proportion at Google, which is 37%. These build failures hinders a project's development process so that they need to be fixed as soon as possible. However, many developers do not have the required expertise to repair build scripts [34]. Therefore, automatic repair of build scripts can be desirable for software project managers and developers.

Automatic generation of software patches is an emerging technique, and has been addressed by multiple previous research efforts. For example, GenProg [14] and PAR [20] achieve promising result for automatic bug fixing. But these works are designed for repairing source code written in different programming languages. In contrast, repairing build scripts has its unique challenges. First, although the code similarity assumption (both GenProg and PAR are taking advantage of this assumption to fetch patch candidates from other portion of the project or other projects) still holds for build scripts, build-script repair often involves open knowledge that do not exist in the current project, such as a newly available version of a dependency or a build-tool plug-in (See Example 1). Second, unlike source code bugs, build failures does not have a test suite to facilitate fault localization [31] and to serve as the fitness function [12]. Third, while different programming languages share similar semantics (so that code patterns / templates can be adapted and reused), the semantics of build scripts are very different from normal programs, so we need to re-develop abstract fix templates for build scripts.

On the other hand, there are also special opportunities we can take advantage of in the repair of build scripts. First, build failures often provide richer log information than normal test failures, and the build failure log can often be used to determine the reason and location of a build failure. Second, build scripts are programs in a

Example 1 Gradle Version Dependency Change *(puniverse/quasar: 2a45c6f)*

```
task wrapper(type: Wrapper) {
- gradleVersion = '1.11'
+ gradleVersion = '2.0'
}
```

specific domain, so it is possible to develop more specific build-fix templates (e.g., involving more domain-specific concepts such as versions, dependencies instead of general concepts like parameters, variables). Third, many build failures are related to the build tools and environments. These failures are not project-specific and may be recursive [39, 51] in different projects, so fix patterns can often be used beyond a project's boundary.

In this paper, we propose a novel approach, HireBuild, to generate patches for build scripts. Our insight is that, since many software projects use the same build tool (e.g., Gradle), similar build failures will result in similar build logs. Therefore, given a build failure, it is possible to use its build-failure log to locate similar build failures from a historical build-fix dataset, and adapt historical fixes for these new failures. Specifically, our technique consists of the following three phases. First, for a given build failure, based on build log similarity, we acquire a number of historically fixed build failures that have the most similar build logs. We refer to these build fixes as seed fixes. Second, from build-script diffs of the seed fixes, we extract a number of fix patterns based on our predefined fix-pattern templates for build scripts, and rank the patterns by their commonality among seed fixes. To generate build-script diffs, our approach uses an existing tool GumTree [11] which extracts changes of Java source code, XML, JavaScript code change. Third, we combine the patterns with information extracted from the build scripts and logs of the build failure to generate a ranked list of patches, which are applied to the build script until the build is successful.

Although following the general generation-validation process for program repair, our technique is featured with following major differences to address the challenges and take advantage of the opportunities in build-script repair.

- **Build log analysis.** Build logs contain a lot of information about the location and reason of build failures, and sometimes even provide solutions. Our build log analysis parses build logs and extracts information relevant to build failures. Furthermore, HireBuild measures the similarity of build logs based on extracted information.
- **Build-fix-pattern templates.** There are a number of common domain-specific operations in build scripts, such as including / excluding a dependency, updating version numbers, etc. In HireBuild, we developed build-fix-pattern templates to involve these common operations specific to software build process.
- **Build validation.** In build-script repair, without test cases, we need a new measurement to validate generated patches. Specifically, we use the successful notification in the build log and the numbers of compiled source files to measure build successfulness.

In our work, we focus on repair of build scripts, so we do not consider compilation errors or unit-testing failures (although they also cause build failures) as they can be easily identified based on build logs and may be automatically repaired with existing bug repair techniques. Furthermore, we use Gradle (based on Groovy) as our targeted build tool as it is the most promising Java build tools now, and recent statistics [40] show that more than 50% of top GitHub apps have already switched to Gradle.

In our evaluation, we extracted 175 reproducible build fixes with corresponding build logs and build script changes from Travistorrent dataset [9] on February 8, 2017, the build fixes are from 54 different projects). To evaluate HireBuild, we use the earlier 135 build fixes as our training set, and 40 later actual build failures (chronologically 135 earlier and 40 later bug fixes among the 175 regardless of which project they belong to) as our evaluations set. Among these 40 build failures, we reproduced 24 build failures in our test environment. Empirical evaluation results show that our approach is able to generate a fix for 11 of the 24 reproduced build script failures which gives same build output as developers' original fix. Overall, our work presented in the paper makes the following contributions.

- A novel approach and tool to automatic patch generation for build scripts to resolve software build failures.
- A dataset of 175 build fixes which can serve as the basis and a benchmark for future research.
- An empirical evaluation of our approach on real-world build fixes.
- An Abstract-Syntax-Tree (AST) diff generation tool for Gradle build scripts, which potentially have more applications.

The remaining part of this paper is organized as follows. After presenting a motivation example of how build-script repair is different from source-code repair in Section 2, we describe the design details of HireBuild in Section 3. Section 4 presents the evaluation of our approach, while Section 5 presents discussion of important issues. Related works and Conclusion will be discussed in Section 6 and Section 7, respectively.

2 MOTIVATING EXAMPLE

In this section, we introduce a real example from our dataset to illustrate how patch generation of build scripts is different from patch generation of source code. Example 2 shows a build failure and its corresponding patch where the upper part shows the most relevant snippet in the build-failure log and the lower part shows the code change to resolve the build failure. The project name and commit id are presented after the example title.

In this build failure, the build-failure log complains that there are two conflicting versions of `slf4j` module, and the bug fix is to add an exclusion of the module in the compilation of `Galaxy` component. Although this build fix is just a one-line simple fix, it illustrates differences between source-code repair and build-script repair in the following aspects.

First, it is possible to find from existing scripts or past fixes that we need to perform an `exclude` operation, however, since `org.slf4j` never appears in the script (it is transitively referred and will be downloaded from Gradle central dependency repository at runtime), the string "org.slf4j" can be hard to generate, and enumerating all possible strings is not a feasible solution. The string can actually be generated by comparing the build-failure log and available modules in Gradle central dependency repository, but this is very different

Example 2 A Gradle Build Failure and Patch *(puniverse/quasar: Build Failure Version:017fa18, Build Fix Version:509cd40)*

```
Could not resolve all dependencies for
    configuration ':quasar-galaxy:compile'.
> A conflict was found between the following
    modules:
  - org.slf4j:slf4j-api:1.7.10
  - org.slf4j:slf4j-api:1.7.7
```

```
compile ("co.paralleluniverse:galaxy:1.4") {
    ...
    exclude group: 'com.google.guava', module: '
        guava'
 + exclude group: "org.slf4j", module: '*'
      }
```

from source-code patching where all variable names to be referred to are already defined in the code (in the case when a generated fix contains a newly declared variable, the variable can have any name as long as it does not conflict with existing names in the scope).

Second, in build-script repair, we are able to, and need to consider build-specific operations. For example, we should not simply deem `exclude` as an arbitrary method name, but needs to involve its semantics into fix-pattern templates, so that we know a module name will follow the `exclude` command.

Third, the build log information is very important in that it not only provides the name of conflicting dependency, but also provides the compilation task performed when build failure happens, which can largely help patch generation tool to locate the build failure and determine where to apply the patch.

3 APPROACH

The overall goal of HireBuild is to generate build-script patches that can be used to resolve build failures. HireBuild achieves these goals with three steps: (1) log similarity calculation to find similar historical build fixes as seed fixes, (2) extraction of build-fix patterns from seed fixes, and (3) generation and validation of concrete patches for build scripts. In the following subsections, we first introduce preliminary knowledge on Gradle, and then describe the three steps of HireBuild with more details in the following subsections.

3.1 Gradle Build Tool

Gradle is a general purpose build management system based on Groovy and Kotlin [2]. Gradle supports the automatic download and configuration of dependencies or other libraries. It supports Maven and Ivy repositories for retrieving these dependencies. This allows reusing the artifacts of existing build systems.

A Gradle build may consist of one or more build projects. A build project corresponds to the building of the whole software project or a submodule. Each build project consists of a number of tasks. A task represents a piece of work during the building process of the build project, e.g., compile the source code or generate the Javadoc. A project using Gradle describes its build process in the `build.gradle` file. This file is typically located in the root folder of the project. In this file, a developer can use a combination of declarative and imperative statements in Groovy or Kotlin code. This build file defines a project and its tasks, and tasks can also be created and extended dynamically at runtime. Gradle is a general purpose build system hence this build file can perform any task.

3.2 Log Similarity Calculation to Find Similar Fixes

One of the most important characteristic of build script repair is that, a lot of software projects use the same build tools (e.g., Gradle), so that build-failure logs of different projects and versions often share the same format and output similar error messages for similar build errors. So given a new build failure, HireBuild measures the similarity between its build-failure log and the build-failure logs of historical build failures to find its most similar build failures in history dataset.

3.2.1 Build Log Parsing. Gradle build logs typically contain thousands of lines of text. Gradle prints these lines when performing different tasks such as downloading dependencies, compiling source files, and when facing errors during the build. Our point of interest is the error-and-exception part, which typically accounts for only a small portion of the build log. So if we use the whole build log to calculate similarity, the remaining part will bring a lot of noises to the calculation (e.g., build logs from projects that have similar dependencies may be considered similar).

Therefore, we use only the error-and-exception part of the build log to calculate similarity between build logs. An example of the error-and-exception part in Gradle build log is presented as below.

```
* What went wrong:
A problem occurred evaluating project ':android-
    rest'.
>
Gradle version 1.9 is required. Current version is
     1.8. If using the gradle wrapper, try editing
     the distributionUrl in /home/travis/build/47
    deg/appsly-android-rest/gradle/wrapper/gradle-
    wrapper.properties to gradle-1.9-all.zip
```

To extract the error-and-exception part, HireBuild extracts the portion of the build log after the error indicating header in Gradle (e.g., "* What went wrong"). HireBuild extracts only the last error, as the earlier ones are likely to be errors that are tolerated and are thus not likely to be the reason for the build failure. Furthermore, when there are exception stack traces in the error-and-exception part, HireBuild removes the stack traces for two reasons. First, stack traces are often very long, so they may dominate the main error message and bring noise (as mentioned above). Second, stack traces are often different from project to project so they cannot catch the commonality between build failures.

3.2.2 Text Processing. After we extracted the error-and-exception part from the build-failure log, we perform the following processing to convert the log text to standard word vector.

- `Text Normalization` breaks plain text into separate tokens and splits camel case words to multiple words.

- `Stop word Removal` removes common stop words, punctuation marks etc. For better similarity, HireBuild also removes common words for building process including "build", "failure", and "error".
- `Stemming` is the process of reducing inflected words to their root word. As an example, the word "goes" derived from word "go". The stemming process converts "goes" to its root word "go". For stemming we applied popular Porter stemming algorithm [47].

3.2.3 Similarity Calculation. With the generated word vector from the error-and-exception part of the build-failure logs, we use the standard Term Frequency–Inverse Document Frequency (TF–IDF) [33] formula to weight all the words. Finally, we calculate cosine similarity between the log of build failure to be resolved and all build-failure logs of historical build fixes in our training set, and fetch the most similar historical fixes. HireBuild uses the five most similar historical fixes as seed fixes to generate build-fix patterns.

3.3 Generation of Build-Fix Patterns

To generate build-fix patterns, for each seed fix, HireBuild first calculates the code difference between the versions before and after the fix. The code difference consists of a list of elementary revisions including insertions, deletions and updates. Then, for each revision, HireBuild generalizes it to hierarchical pattern and merges similar patterns. Finally, HireBuild flattens the hierarchical pattern to generate a set of build-fix patterns, and ranks these patterns.

3.3.1 Build-Script Differencing. In this phase, for each seed fix, we extract Gradle build script commits before and after fix, and convert the script code to AST representation. Gradle build uses Groovy [2]-based scripting language extended with domain-specific features to describe software build process. With support of the Groovy parser, AST representation of script code can be generated.

Our goal is to generate an abstract representation of code changes between two commits. Having build script content represented as an AST, we can apply tree difference algorithms, such as ChangeDistiller [13] or GumTree [11], to extract AST changes with sufficient abstraction. In particular, HireBuild uses GumTree to extract changes between two Gradle build scripts. GumTree generates a diff between two ASTs with list of actions which can be insertion, deletion, update, and movement of individual AST nodes to transfer from a source version to a destination version. However, GumTree generates a list of AST revisions without node type information, so we revise GumTree to include the information. Furthermore, HireBuild also records the ancestor AST nodes of the changed AST subtree. Such ancestor AST nodes are typically the enclosing expression, statement, block, and task of the change, and they are helpful for merging different seed fixes for more general patterns, and for determining where the generated patches should be applied. As mentioned earlier, in Gradle scripts, a task is a piece of work which a build performs, and a script block is a method call with parameters as closure [3], so keeping such information helps to apply patches to a certain block or task. Example 3 shows an exemplar output of HireBuild's build-script differencing module, in which the operation, node type, and ancestor nodes are extracted. Note that HireBuild extracts only one level of parent expression to avoid potential noises. As shown in the example, the task/block name can be empty if the fix is not in any tasks/blocks.

Example 3 Build Script Differencing Output *(BuildCraft/BuildCraft: 98f7196)*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<patch>
<lineno id="30"><exp id="0">
   <operation>Update</operation>
   <nodetype>ConstantExpression</nodetype>
   <nodeexp>1.7.2-10.12.1.1079</nodeexp>
   <nodeparenttype>BinaryExpression</nodeparenttype>
   <nodeparentexp>(version = 1.7.2-10.12.1.1079)
   </nodeparentexp>
   <nodeblockname>minecraft</nodeblockname>
   <nodetaskname> </nodetaskname></exp>
 </lineno>
</patch>
```

3.3.2 Hierarchical Build-Fix Patterns. In some rare cases we can directly use the concrete build-fix pattern to generate a correct patch. Example 4 provides such a patch from `Project: nohana/Laevatein:a2aaca4`. There exists an exactly same build fix in the training set (from a different project).

Example 4 Training Project Fix *(journeyapps/zxing-android-embedded: 12cfa60)*

```
+ lintOptions {
+ abortOnError false
+ }
```

However, in more common scenarios, code diffs generated from seed fixes are too specific and cannot be directly applied as patches. Consider Examples 5 and 6, changes made in different project are similar, but if we consider concrete change of Example 5 as "Update 1.7.2-10.12.1.1079" then this change can hardly be applied to other scripts. Therefore, we need to infer more general build-fix patterns from them.

Example 5 Gradle Build Fix *(BuildCraft/BuildCraft: 98f7196)*

```
- version = "1.7.2-10.12.1.1079"
+ version = "1.7.2-10.12.2.1121"
```

Example 6 Gradle Build Fix *(ForgeEssentials/ForgeEssentialsMain:fcbb468)*

```
-version = "1.4.0-beta7"
+version = "1.4.0-beta8"
```

Specifically, HireBuild infers a hierarchy of build-fix patterns from each seed fix by generalizing each element in the differencing output of the seed fix. For example, the hierarchies generalized from Examples 5 and 6 are shown in Figure 1. From the figure, we can see that, HireBuild does not generalize operations and the node type of expression that are involved in the fix (i.e., ConstantExpression), because a change on those typically indicates a totally different fix. HireBuild also does not include the task and block information in

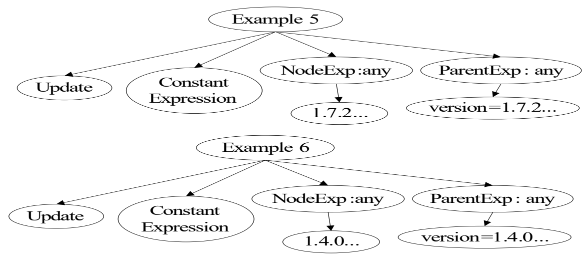

Figure 1: Hierarchies of Build-Fix Patterns

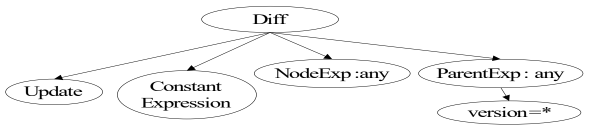

Figure 2: Merged Hierarchies

the pattern as they are typically not a part of the fix. Given a hierarchy, by choosing whether and which leaf node to be generalized, we can generate patches at different abstract levels. For example, if we generalize the parent expression from `version=1.7.2...` to `ParentExp: any`, we generate a pattern that updates a value `1.7.2...` without considering its parent. If we generalize both the parent expression and the node expression, we generate a pattern that update any constants in the script. Note that HireBuild does not consider the cases where the node expression is generalized but the parent expression is not, as such a pattern can never match real code.

3.3.3 Merging Build-Fix Patterns. After generating hierarchies of build-fix patterns, HireBuild first tries to merge similar hierarchies. For example, the two hierarchies in Figure 1 will be merged to a hierarchy shown in Figure 2. HireBuild merges only a pair of hierarchies with the same operation and node type (Update and Constant Expression in this case). During the merging process, HireBuild merges hierarchies recursively from their root node, and merges nodes with exactly the same value. If two nodes to be merged have different constant values, HireBuild does not merge them and their children nodes. If two nodes to be merged have different expression values, HireBuild extracts their corresponding AST tree, and merges the AST tree so that the common part of the expressions can be extracted. In Figure 2, since the expressions `version=1.7.2...` and `version=1.4.0...` share the same child nodes `version` and `=`, a node `version=*` is added. Note that more than two hierarchies can be merged in the same way if they share the same operation and node type.

3.3.4 Ranking of Build-Fix Patterns. After hierarchies are merged, HireBuild calculates frequencies of build-fix patterns among seed fixes. If a hierarchy cannot be merged with other hierarchies, all the build-fix patterns in it have a frequency 1 among seed fixes, as they are specific to the seed fix they are from. In a merged hierarchy, the frequency of all its patterns is always the number of original hierarchies being merged. After calculating the frequencies of all build-fix patterns from hierarchies, HireBuild ranks build-fix patterns according to the frequency. For each build-fix pattern α, we counted n^t_α: α's frequency among seed fixes. Then probability of α is as follows.

$$P_\alpha = \frac{n^t_\alpha}{N}$$

where N is the total occurrences of build-fix patterns. Then, we rank the fix patterns based on the probability so that we use higher ranked build-fix patterns first to generate concrete patches. When there are ties between pattern A and B, if A is a generalization of B (A is generated by generalizing one or more leaf nodes of B), we rank B over A. The reason is that, when a build-fix pattern is generalized, it can lead to a larger number of concrete patches (e.g., update `gradleVersion` from any existing version to another existing version), so HireBuild needs to perform more build trials to exhaust all possibilities. As an example, all build-fix patterns from the hierarchy in Figure 2 have the same popularity, but the most concrete pattern: `update constant expression with parent expression version= *` will be ranked highest.

If there is no generalization relation between patterns, HireBuild ranks higher the build-fix patterns from the seed fix with higher ranking (the seed fix whose build failure log is more similar to that of the build failure to be fixed).

3.4 Generation and Validation of Concrete Patches

Before generation of concrete patches, we need to first decide which `.gradle` file to apply the fix. HireBuild uses a simple heuristic, which always choose the first `.gradle` file mentioned in the error part extracted from the build failure log. If no `.gradle` file is mentioned, HireBuild uses the `build.gradle` file in the root folder.

Given a build-fix pattern, and the buggy Gradle build script as input, to generate concrete patches, HireBuild first parses the buggy Gradle build script to AST, and then HireBuild tries to find where a patch should be applied.

For updates and deletions, HireBuild matches the build-fix patterns to nodes in the AST. For example, the build-fix pattern `update constant expression with parent expression version = *` can be mapped to an AST node of type `ConstantExpression` and its parent expression node has a value matching `version=*`. When a build-fix pattern can be mapped to multiple AST nodes (very common for general build-fix patterns), and HireBuild generates patches for all the mapped AST nodes. The only exception is when a build-fix pattern is mapped to multiple AST nodes in one block. In build scripts, within the same block, the sequence of commands typically does not matter, so HireBuild retains only the first mapped node in the block to reduce duplication.

For insertions, it is impossible to map a build-fix pattern to an existing AST node, so HireBuild matches the block and task names of the build-fix patterns to the buggy build script. When a build-fix pattern is generated from a hierarchy merging multiple seed-fixes, HireBuild considers the task and block names of all seed-fixes. If a task or block name in the buggy script is matched, HireBuild inserts the build patch at the end of the task or block.

After HireBuild determines which build-fix pattern to apply and where to apply, we finally need to concentrate on the abstract parts of the build-fix pattern and determine the values of the abstract nodes (e.g., value of "*" in the pattern `update constant expression with parent expression version= *`). The most commonly used values in build scripts are (1) identifiers including task names, block names, variable names, etc.; (2) names of Gradle plug-ins and third-party tools / libraries; (3) file paths within the project; and (4) version numbers. HireBuild first determines which type the value to added belongs to, based on the concrete values and AST nodes in the seed fixes leading to the build-fix pattern to be applied. HireBuild identifies version number and file paths based on regression expression matching (e.g., HireBuild can determine that `1.4.0-beta8` is a version number), and task / block / variable names by scanning the AST containing the seed fix. Other types of values including dependencies / plug-in names, and file paths are all specific to certain AST nodes so that they can be easily identified. Once the value type is determined, HireBuild generates values differently for different types as follows.

- Identifiers: HireBuild considers identifiers in the concrete seed fixes, as well as all available identifiers at the fix location.
- Names of plug-ins / libraries / tools: HireBuild considers names appearing in the concrete seed fixes, in the build failure log, and in the buggy build script.
- File paths: HireBuild considers paths appearing in the concrete seed fixes, in the build failure log, and in the buggy build script.
- Version numbers: HireBuild first locates the possible tools / libraries / plug-ins the version number is related to. This is done by searching for all occurrences of the version variable or constant in the AST of the buggy script. Once the tool / library / plug-in is determined, HireBuild searches Gradle central repository for all existing version numbers.

After the build-fix pattern, the location, and the concrete value are determined, a concrete patch is generated and added to the list of patches.

3.4.1 Ranking of Generated Patches. The previous steps generate a large number of patch candidates, so ranking of them is necessary to locate the actual fix as soon as possible. HireBuild ranks concrete patches with the following heuristics. Basically, we give higher priority to the patches which involve values or scopes more similar to the buggy script and the build-failure log.

(1) Patches generated from higher ranked build-fix patterns are ranked higher than those generated from lower ranked build-fix patterns. The initial priority value of a patch is the probability value of its build-fix pattern.
(2) If a patch p is to be applied to a location L, and p is generated from a build-fix pattern hierarchy merged from seed fixes A_1, ..., A_i, ..., A_n. If L resides in a task / block whose name is the same as the task / block name in any A_i, HireBuild adds p's priority value by 1.0.
(3) If a patch involves a value (any one of the four types described in Section 3.4) which appears in the build-failure log. HireBuild adds the priority value of the patch by 1.0.
(4) Rank all patches with updated priority values.

Note that, since the initial priority value is from 0 to 1, in the heuristics, we always add the priority value by 1.0 when certain condition meets, so that it go beyond all the other patches which do not satisfy the condition, no matter how high the initial priority value is.

3.4.2 Patch Application. After the ranked list of patches are generated, HireBuild applies the patches one by one until a timeout threshold is reached or the failure is fixed. HireBuild determines the failure is fixed if (1) the build process returns 0 and the build log shows build success, and (2) all source files that are compiled in the latest successfully built version are compiled if they are not deleted in between. We add the second criterion so that HireBuild can avoid trivial incorrect fixes such as changing the task to be performed from compile to clean up and to eliminate fake patches. HireBuild stops applying patches after it reaches the first patch passing the patch validation. Though there may be multiple valid patches, we apply only the first one that passes the validation.

HireBuild generally focuses on one line fixes as most other software repair tool does. But it also includes a technique to generate multi-line patches if the failure is not fixed until all single line patches are applied. Multi-line patches can be viewed as a combination of single line patches, but it is impossible to exhaust the whole combination space. Example 7 shows a bug fix, which can be viewed as the combination of three one-line patches (two deletions and one insertion). To reduce the search space of patch combination, HireBuild considers only the combination that occurs in original seed fixes. Consider two one-line patches A and B, which are generated from hierarchies HA and HB. HireBuild considers the combination (A, B) only if HA and HB can be generalized from a same seed fix. After the filtering, HireBuild ranks patch combinations by the priority sum of the patches in the combination.

Example 7 Template with abstract node fix *(passy/Android-DirectoryChooser:27c194f)*

```
dependencies {
...
 - testCompile
     files('testlibs/robolectric-2.4-SNAPS
     HOT-jar-with-dependencies.jar')
 - androidTestProvided
     files('testlibs/robolectric-2.4-SNAPS
     HOT-jar-with-dependencies.jar')
 + androidTestCompile
     'org.robolectric:robolectric:2.3+'
...
}
```

4 EMPIRICAL EVALUATION

In this section, we describe our dataset construction in Section 4.1 and our experimental settings in Section 4.2, followed by research questions in Section 4.3 and experiment results in Section 4.4. Finally, we discuss the threats to validity in Section 4.5.

4.1 Dataset

We evaluate our approach to build-script repair on a dataset of build fixes extracted from the TravisTorrent dataset [9] snapshot at February 8, 2017. The tool and bug set used in our evaluation are all available at our website [1]. TravisTorrent provides easy-to-use Travis

[1] HireBuild Dataset and Tools: https://sites.google.com/site/buildfix2017/

Table 1: Dataset Summary

Type	Count
# Total Number of Projects	54
# Maximum Number of Fix From Single Project	25
# Minimum Number of Fix From Single Project	1
# Average Number of Fix Per Project	3.2
# Total Number of Fix	175
# Training Fix Size	135
# Testing Fix Size	40
# Reproducible Build Failure Size for Testing	24

CI build data to the masses through its open database. Though it provides large amount of build logs and relevant data, our point of interest is build status transition from error or fail status to pass status with changes in build scripts. From the version history of all projects in the TravisTorrent dataset, we identified as build fixes the code commits that satisfy: (1) the build status of their immediate previous version is fail / error; (2) the build status of the committed version is success; and (3) they contain only changes in gradle build scripts. Since HireBuild focuses on build script errors, we use code commits with only build-script changes so that we can filter out unit test failures and compilation failures. Our dataset may miss the more complicated build fixes that involve a combination of source-code changes and build-script changes, or a combination of build-script changes from different build tools (e.g., Gradle and Maven). HireBuild currently does not support the generation of such build fixes cross programming languages. Actually, fixing such bugs are very challenging and is not supported by any existing software repair tools.

From the commit history of all projects, we extracted a dataset of 175 build fixes. More detailed information about our data set is presented in Table 1. We can see that these fixes are from 54 different projects, with maximal number of fixes in one project to be 25.

We ordered the build fixes according to the code commit time stamp, and use 135 (75%) earlier build fixes as the training set and the rest 40 build fixes (25%) as the evaluation set. **Therefore, all the build fixes in our evaluation set are chronically later than the build fixes in our training set.** Note that we combine all the projects in both training sets and evaluation sets, so our evaluation is cross-project in nature.

Among these 40 build fixes for evaluation, we successfully reproduced 24 build failures. The remaining 16 build failures cannot be reproduced in our test environment for the following three reasons: (1) a missing library or build configuration file was originally missing from the central repository and caused the build failure, but they are added later; (2) a flawed third-party library or build configuration caused the build failure, but the flaws are fixed and flawed releases are no longer available on the Internet; and (3) the failure can be reproduced only with specific build commands and options which are not recorded in the repository. For case (3), we were able to reproduce some bugs by trying common build command options. We also contacted the TravisCI people about the availability of such commands / options, but they could not provide them to us. For training set, we did not reproduce build failures since we trust the software version history in TravisTorrent that human made changes resolved the build failures and we extracted only seed fixes from training set.

Table 2: Project-wise Build Failure / Fix List

Project Name	**#Failures**	**#Correctly Fixed**
aol/micro-server	2	1
BuildCraft/BuildCraft	2	0
exteso/alf.io	1	1
facebook/rebound	1	1
griffon/griffon	1	0
/btrace	1	1
jMonkeyEngine/jmonkeyengine	2	0
jphp-compiler/jphp	1	0
Netflix/Hystrix	2	0
puniverse/quasar	6	2
RS485/LogisticsPipes	5	5
Total	24	11

4.2 Experiment Settings

TravisTorrent datset provides Travis CI build analysis result as SQL dump and CSV format. We use SQL dump file for our experiment. We use a computer with 2.4 GHz Intel Core i7 CPU with 16GB of Memory, and Ubuntu 14.10 LTS operating system. We use MySQL Server 5.7 to store build fix changes. In our evaluation, we use 600 minutes as the time out threshold for HireBuild.

4.3 Research Questions

In our research experiment, we seek to answer following research questions.

- **RQ1** How many reproducible build failures in the evaluation set can HireBuild fix?
- **RQ2** What are the amount of time HireBuild spends to fix a build failure?
- **RQ3** What are the sizes of build fixes that can be successfully fixed and that cannot be fixed?
- **RQ4** What are the reasons behind unsuccessful build-script repair?

4.4 Results

RQ1: Number of successfully fixed build failures. In our evaluation, we consider a fix to be correct only if there is no build failure message in build log after applying patch, and the build result (i.e., all compiled classes) are exactly the same as those generated by the manual fix. Among 24 reproducible build failures in the test set, we can generate the correct fix for 11 of them. Table 2 shows the list of projects that are used for testing. Columns 2 and 3 represent the number of build failures and the number of those fixed successfully.

Figure 3 shows the breakdown of successful build fixes according the type of changes. With HireBuild, we can correctly generate 3 fixes about gradle option changes, 3 fixes about property changes, 2 fixes about dependency changes and external-tool option changes, respectively, and 1 fix about removing incompatible statements. Example 8 shows a build fix that is correctly generated by HireBuild falling in the category of external-tool option changes. The build failure is caused by adding a new option which is compatible only with Java 8. So the fix is to add an if condition to check the Java version. Note that this fix involves applying the combination of two insertion patches, but HireBuild still can fix it as there are a seed fix that contains both build-fix patterns.

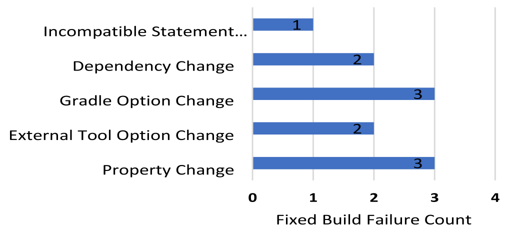

Figure 3: Breakdown of Build Fixes

Example 8 A Build Fix Correctly Generated By HireBuild *(puniverse/-quasar:33bb265)*

```
+if(JavaVersion.current().isJava8Compatible()){
 +tasks.withType(Javadoc) {
 options.addStringOption('Xdoclint:none', '-
    quiet')
 + }
 + }
```

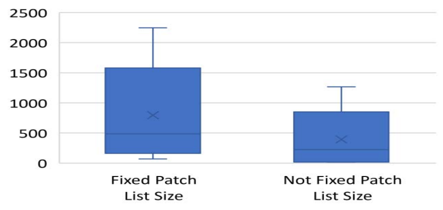

Figure 4: Patch List Sizes

RQ2: Time Spent on Fixes Time spent on fixes is very important for build failures as they need to be fixed timely. The size of patch list has impact on automatic build script repair. If patch list size is too large, it will take large time span to generate fix sequence. We compare patch list size of build failures we can correctly fix and patch list size of build failures we cannot correctly fix, and present the result in Figure 4. From the figure, we can see that for fixed build failures, the patch list has minimum size of 68 and maximum size of 2,245, while median is 486. For non-fixed build failures, patch list minimum, median and maximum are 8, 223 and 1,266 respectively, which are lower than fixable build failure's patch list. The reason behind this result is that for non-fixable build failure, HireBuild cannot find similar build fixes in the training set, and thus the generated build-fix patterns cannot be easily mapped to AST nodes in buggy scripts.

For the 11 fixed build failures, we compared in Figure 5 the time HireBuild spent on automatic fixing build failures with the manual fix time of the build failures in the commit history. Manual fix time comes from commit information and we use the time difference between build failure inducing commit and build failure fix commit as the manual fix time. From Figure 5, we can see that build script fix generated by our approach takes minimum 2 minutes, maximum 305 minutes and median value of 44 minutes. While human fix takes minimum less than one minute, maximum 5,281 minutes and median 42 minutes. We can see that for the fixable build failures, HireBuild fixed them with time comparable to manual fixes.

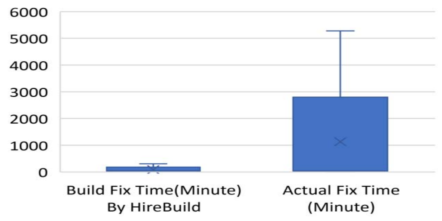

Figure 5: Amount of Time Required for Build Script Fix

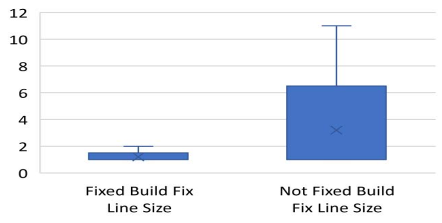

Figure 6: Actual Fix Sizes

Table 3: Cause of unsuccessful patch generation

Fix Type	#of Failures
Project specific change adaption	2(15%)
No matching patterns	6(46%)
Dependency resolution failures	3(23%)
Multi-location fixes	2(15%)

RQ3: Actual Fix Size. Patch size has impact on automatic program repair. According to Le et al. [21], bugs with over six lines of fix are difficult for automatic repair. In our dataset we have not performed any filtering based on actual fix size. But during result analysis, we performed statistical analysis to find out the sizes of build-fix patches, and the difference in size between the patches our approach can correctly generate and the patches our approach cannot. According to Figure 6, fixed build script failures contain minimum one, maximum two and median one. Actually 9 of the fixes contain only 1 statement change, and 2 of the fixes contain only 2 statement changes. For non-fixed Gradle build script failures minimum change size is one, while maximum and median change size is 11 and 1 respectively. Therefore, our approach mainly works in the cases where the number of statement changes is small (1 or 2), which is similar to other automatic repair tools.

RQ4: Failing reasons for the rest 13 build failures. For 54.16% of evaluated build failures, our approach cannot generate build fix. So, we performed manual analysis to find out why our approach fails. We check whether the reason is related to generation of version numbers, dependency names, etc. Then we categorize these failure reasons to four major groups: (1) Project specific change adaption, (2) Non-matching patterns, (3) Dependency resolution failures, and (4) Multi-location fixes, as shown in Table 3.

Project specific change adaption indicates those changes that are dependent on project structure, file path etc. As build script manages build and its configuration, so there are project specific change issues and with our approach we cannot adapt the build-fix patterns. Example 9 shows such a build fix where it uses a specific path in build script.

Non-matching patterns indicates that our automatic patterns generation failed to provide required pattern that can resolve the build

Example 9 Project specific change *(Netflix/Hystrix:6600947)*

```
+ if ( dep.moduleName == 'servlet-api' ) {
+ it.artifactId[0].text() == dep.moduleName
    &&
+ asNode().dependencies[0].dependency.find{
+ ...
+ }}
+ }
```

failure. HireBuild could not generate appropriate patterns for 6 failures which account 46% of failures. This may be due to limited size of training data and insufficient number of available build fixes that we used for template generation.

Dependency resolution failures happens for some project when HireBuild did not find an actual dependency from central repository based on build log error. Even if we find dependency based on miss-compiled classes, that may not match with actual fixing. Example 10 shows such a dependency update where our approach failed to generate the dependency name.

Example 10 Dependency resolve Issue *(BuildCraft/BuildCraft:12f4f06)*

```
 - mappings = 'snapshot_20160214'
 + mappings = 'stable_22'
```

Multi-Location Fixes happen when we need to apply multiple patches to fix a single build failure. HireBuild considers only limited combinations of patches as introduced in Section 3.4.2. Example 11 shows such a case where our patch generation technique generated the two "exclude" statements in two different patches. But this build failure is fixed only when we apply both "exclude" statement change simultaneously.

Example 11 Dependency resolve Issue *(joansmith/quasar:64e42ef)*

```
-jvmArgs ' -Xbootclasspathp:
  ${System.getProperty(üser.home)}jsr166.jar'
-testCompile 'org.testng:testng:6.9.6'
+testCompile('org.testng:testng:6.9.6') {
 +exclude group:'com.google.guava',
     module:'*'
 +exclude group:'junit', module:'*'
 +}
```

4.5 Threats of Validity

There are three major threats to the internal validity of our evaluation. First, there can be mistakes in our data processing and bugs in the implementation of HireBuild. To reduce this threat, we carefully double checked all the code and data in our evaluation. Second, the successful fixes generated by HireBuild may still have subtle differences with the manual fixes. Furthermore, the manual fixes that we use as the ground truth may in itself has flaws. To reduce this threat, we used a strict criterion for correct fixes. We need the automatically generated fix to generate exactly the same build results as those generated by the manual fix. Third, the manual fixing time collected in the commit history may be longer the actual fixing time as developers may choose to wait and not to fix the bug. We agree that this can happen but we believe the difference is not large as developers typically want to fix failure as soon as possible so that it does not affect other developers.

The major threat to the external validity of our evaluation is that we use a evaluation set with limited number of reproducible bug fixes. Furthermore, our evaluation set contains only build fixes where only Gradle build scripts are changed. So it is possible that our conclusion is limited to the data set, Gradle-script fixes, or Gradle-script-only fixes. Figure 3 shows that our evaluation set already covers a large range of change types, and we plan to expand our evaluation set to more build failures and reproduce more bugs as TravisTorrent data set grows overtime. Gradle is the most widely adopted building system now, and its market share is still increasing. We also did a statistics on the number of build fixes with both Gradle-script changes and other file-changes and found 263 of them. Compared with 175 build fixes in our dataset, Gradle-script-only fixes accounts for a large portion of build script fixes for Gradle systems.

5 DISCUSSION

Patch Validation and Build Correctness. In the patch validation stage, HireBuild deems a patch as valid if applying it results in a successful build message, and all the files that are compiled in the most recent successful build are still compiled as long as they are not deleted. Our evaluation uses a more strict constraint which requires the compiled files in the automatically fixed build to be exactly the same as those in the manually fixed build. The evaluation results show that our patch validation strategy is very effective because in all 11 fixed builds in our evaluation, the first patches passing validation are confirmed to be correct patches. The reason is that, based on the same compiler, once a source file is successfully compiled, it is unlikely to be compiled in different ways. The only exception is that a library-class reference is resolved to a wrong class when a wrong dependency is added. Furthermore, to pass compilation, the wrong class must accidentally have compatible behaviors (e.g., methods) with the correct class. Such coincidence is not likely to happen.

Build Environment. Build environment defines the environment of a system that compiles source code, links module and generates assembles. From developer's point of view, they install all required dependencies like Java, GCC and other frameworks. But when projects are built in different environment then build problem can be generated. For example, if certain project has dependency on Java 1.8 then building the project in build environment with Java 1.7 might generate build failure. This a challenge for build automation as well as automatic build repair. During software evaluation, developers change environment dependency based on functional requirements or efficiency. With version changes, developers build the software having those changed dependencies. But for build script repair, if we change the version of any dependency and keep the build environment as it was before, then the fix might not resolve build failures. For Android projects environment, this issue creates greater impact as in most Gradle build script it mentions SDK version, build tool version etc. inside build script. As a result, build script version dependency and build environment should be synced to avoid build breakage.

6 RELATED WORK

6.1 Automatic Program Repair

Automatic program repair is gaining research interest in the software engineering community with the focus to reduce bug fixing time and effort. Recent advancements in program analysis, synthesis, and machine learning have made automatic program repair a promising direction. Early software repair techniques are mainly specific to predefined problems [19, 37, 45, 50]. Le Goues et al. [14] GenProg which is one of the earliest and promising search based automatic patch generation technique based on genetic programming. Patch generated by this approach follows random mutation and use test case for the verification of the patch. Later in 2012, authors optimized their mutation operation and performed systematic evaluation 105 real bugs [22]. RSRepair [32] performs similar patch generation based on random search. D. Kim et al. [20] proposed an approach to automatic software patching by learning from common bug-fixing patterns in software version history, and later studied the usefulness of generated patches [43]. AE [46] uses deterministic search technique to generate patch. Pattern-based Automatic Program Repair(PAR) [20] uses manually generated templates learned from human written patches to prepare a patch. PAR also used randomized technique to apply the fix patches. Nguyen et al. [30] proposed SemFix, which applied software synthesis to automatic program repair, by checking whether a suspicious statement can be re-written to make a failed test case pass. Le et al. [21] mines bug fix patterns for automatic template generation and uses version control history to perform mutation. Prophet [24] proposed a probabilistic model learned from human written patched to generate new patch. The above mentioned approaches infers a hypothesis that new patch can be constructed based on existing source. This hypothesis also validated by Barr et al.[8] that 43 percent changes can be generated from existing code. With this hypothesis, we proposed first approach for automatic build failure patch generation. Tan and Roychoudhury proposed Relifix [42], a technique that taking advantage of version history information to repair regression faults. Smith et al. [38] reported an empirical study on the overfitting to test suites of automatically generated software patches. Most recently, Long and Rinard proposed SPR [23], which generates patching rules with condition synthesis, and searches for the valid patch in the patch candidates generated with the rules. Angelix [28] and DirectFix [27] both use semantics-based approach for patch generation. To fix buggy conditions, Nopol [49] proposes test-suite based repair technique using Satisfiability Modulo Theory(SMT). Although our fundamental goal is same, but our approach is different than others in several aspects: 1) Our approach is applicable for build scripts, 2) We generate automatic fix template using build failure log similarity, 3) With abstract fix template matching we can generate fix candidate lists with reasonable size.

6.2 Analysis of Build Configuration Files

Analysis of build configuration files is growing as an important aspect for software engineering research such as dependency analysis for path expression, migration of build configuration file and empirical studies. On dependency analysis, Gunter [7] proposed a Petri-net based model to describe the dependencies in build configuration files. Adams et al. [5] proposed a framework to extract a dependency graph for build configuration files, and provide automatic tools to keep consistency during revision. Most recently, Al-Kofahi et al. [6] proposed a fault localization approach for make files, which provides the suspiciousness scores of each statement in a make files for a building error. Wolf et al. proposed an approach [48] to predict build errors from the social relationship among developers. McIntosh et al. [26] carried out an empirical study on the efforts developers spend on the building configurations of projects. Downs et al. [10] proposed an approach to remind developers in a development team about the building status of the project. On the study of building errors, Seo et al. [36] and Hassan et al. [15, 17] carried out empirical studies to categorize build errors. Their study shows that missing types and incompatibility are the most common type of build errors, which are consistent with our findings.

The most closely related work in this category is SYMake developed by Tamrawi et al. [41]. SYMake uses a symbolic-evaluation-based technique to generate a string dependency graph for the string variables/constants in a Makefile, automatically traces these values in maintenance tasks (e.g.,renaming), and detect common errors. Compared to SYMake, the proposed project plans to develop build configuration analysis for a different purpose (i.e., automatic software building). Therefore, the proposed analysis estimates run-time values of string variables with grammar-based string analysis instead of string dependency analysis, and analyzes flows of files to identify the paths to put downloaded files and source files to be involved. On migration of build configuration files, AutoConf [1] is a GNU software that automatically generates configuration scripts based on detected features of a computer system. AutoConf detects existing features (e.g., libraries, software installed) in a build environment, and configure the software based on pre-defined options.

7 CONCLUSION AND FUTURE WORK

For Source code, automatic patch generation research is already in good shape. Unfortunately, existing techniques are only concentrated to source code related bug fixing. In this work, we propose the first approach for automatic build fix candidate patch generation for Gradle build script. Our solution works on automatic build fix template generation based on build failure log similarity and historical build script fixes. For extracting build script changes, we developed GradleDiff for AST level build script change identification. Based on automated fix template we generated a ranked list of patches. In our evaluation, our approach can fix 11 out of 24 reproducible build failures.

In future, we plan to increase training and testing data size for better coverage of build failures with better evaluation and perform study on patch quality for the patches generated by out tool. Moreover, change patterns from general build script commits may also be useful, and we have a plan to work on build script change patterns regardless of build status. Apart from that, we are planning to apply search based technique such as genetic programming with fitness function on our patch list to better rank our generated patches and apply combination of patches.

ACKNOWLEDGEMENTS. The authors are supported in part by NSF Awards CCF-1464425, CNS-1748109, and DHS grant DHS-14-ST-062-001.

REFERENCES

[1] 2015. GNU Autoconf - Creating Automatic Configuration Scripts. http://www.gnu.org/software/autoconf/manual/index.html. (2015). Accessed: 2015-10-25.

[2] 2017. The Gradle build language. https://docs.gradle.org/current/userguide/writing_build_scripts.html. (2017). Accessed: 2017-04-30.

[3] 2017. Gradle Build Script Structure. https://docs.gradle.org/3.5/dsl/. (2017). Accessed: 2017-04-30.

[4] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2007. The Evolution of the Linux Build System. *ECEASST* 8 (2007).

[5] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter. 2007. Design recovery and maintenance of build systems. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. 114–123. DOI: http://dx.doi.org/10.1109/ICSM.2007.4362624

[6] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault Localization for Build Code Errors in Makefiles. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 600–601. DOI: http://dx.doi.org/10.1145/2591062.2591135

[7] Nasreddine Aoumeur and Gunter Saake. 2004. Dynamically Evolving Concurrent Information Systems Specification and Validation: A Component-based Petri Nets Proposal. *Data Knowl. Eng.* 50, 2 (Aug. 2004), 117–173. DOI: http://dx.doi.org/10.1016./j.datak.2003.10.005

[8] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 306–317.

[9] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.

[10] John Downs, Beryl Plimmer, and John G. Hosking. 2012. Ambient Awareness of Build Status in Collocated Software Teams. In *Proceedings of ICSE*. 507–517.

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 313–324. DOI: http://dx.doi.org/10.1145/2642937.2642982

[12] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2010. Designing better fitness functions for automated program repair. In *GECCO*, Martin Pelikan and Jürgen Branke (Eds.). ACM, 965–972.

[13] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (Nov 2007), 725–743. DOI: http://dx.doi.org/10.1109/TSE.2007.70731

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 54–72. DOI: http://dx.doi.org/10.1109/TSE.2011.104

[15] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang. 2017. 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)*. 38–47.

[16] F. Hassan and X. Wang. 2017. Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 157–162.

[17] Foyzul Hassan and Xiaoyin Wang. 2017. Mining readme files to support automatic building of java projects in software repositories: Poster. In *Proceedings of the 39th International Conference on Software Engineering Companion*. 277–279.

[18] Hubert Klein Ikkink. 2015. *Gradle Dependency Management*. Packt Publishing.

[19] Hannes Kegel and Friedrich Steimann. 2008. Systematically Refactoring Inheritance to Delegation in Java. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. 431–440.

[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811. http://dl.acm.org/citation.cfm?id=2486788.2486893

[21] X. B. D. Le, D. Lo, and C. L. Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224. DOI: http://dx.doi.org/10.1109/SANER.2016.76

[22] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 3–13. http://dl.acm.org/citation.cfm?id=2337223.2337225

[23] Fan Long and Martin Rinard. 2015. Staged Program Repair in SPR. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*.

[24] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 298–312.

[25] Shane Mcintosh, Bram Adams, and Ahmed E. Hassan. 2012. The Evolution of Java Build Systems. *Empirical Softw. Engg.* 17, 4-5 (Aug. 2012), 578–608. DOI: http://dx.doi.org/10.1007/s10664-011-9169-5

[26] S. McIntosh, B. Adams, T.H.D. Nguyen, Y. Kamei, and AE. Hassan. 2011. An empirical study of build maintenance effort. In *Software Engineering (ICSE), 2011 33rd International Conference on*. 141–150. DOI: http://dx.doi.org/10.1145/1985793.1985813

[27] S. Mechtaev, J. Yi, and A. Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. DOI: http://dx.doi.org/10.1109/ICSE.2015.63

[28] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. DOI: http://dx.doi.org/10.1145/2884781.2884807

[29] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.

[30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. 772–781.

[31] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 609–620. DOI: http://dx.doi.org/10.1109/ICSE.2017.62

[32] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 254–265. DOI: http://dx.doi.org/10.1145/2568225.2568254

[33] Juan Ramos and others. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*.

[34] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 345–355. DOI: http://dx.doi.org/10.1109/MSR.2017.54

[35] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 724–734. DOI: http://dx.doi.org/10.1145/2568225.2568255

[36] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of ICSE*. 724–734.

[37] S. Sidiroglou and A. D. Keromytis. 2005. Countering network worms through automatic patch generation. *IEEE Security Privacy* 3, 6 (2005), 41–49.

[38] E. Smith, E. Barr, C. Le Goues, and Y. Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*.

[39] Yoonki Song, Xiaoyin Wang, Tao Xie, Lu Zhang, and Hong Mei. 2010. JDF: detecting duplicate bug reports in Jazz. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 315–316.

[40] Matúš Sulír and Jaroslav Porubän. 2016. A Quantitative Study of Java Software Buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016)*. ACM, New York, NY, USA, 17–25. DOI: http://dx.doi.org/10.1145/3001878.3001882

[41] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. SYMake: A Build Code Analysis and Refactoring Tool for Makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 366–369. DOI: http://dx.doi.org/10.1145/2351676.2351749

[42] S. H. Tan and A. Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *International Conference on Software Engineering*.

[43] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 64–74.

[44] Jesse Tilly and Eric M. Burke. 2002. *Ant: The Definitive Guide* (1st ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[45] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2009. Transtrl: An automatic need-to-translate string locator for software internationalization. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 555–558.

[46] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 356–366.

[47] Peter Willett. 2006. The Porter stemming algorithm: then and now. *Program* 40, 3 (2006), 219–223.

[48] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting Build Failures Using Social Network Analysis on Developer Communication. In *Proceedings of ICSE*. 1–11.

[49] J. Xuan, M. Martinez, F. DeMarco, M. ClÃl'ment, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (Jan 2017), 34–55. DOI:http://dx.doi.org/10.1109/TSE.2016.2560811

[50] Hongyu Zhang, Hee Beng Kuan Tan, Lu Zhang, Xi Lin, Xiaoyin Wang, Chun Zhang, and Hong Mei. 2011. Checking enforcement of integrity constraints in database applications based on code patterns. *Journal of Systems and Software* 84, 12 (2011), 2253–2264.

[51] Hao Zhong and Na Meng. 2017. An empirical study on using hints from past fixes: poster. In *Proceedings of the 39th International Conference on Software Engineering Companion*. 144–145.