Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges

Foyzul Hassan[1], Shaikh Mostafa[1], Edmund S. L. Lam[2] and Xiaoyin Wang[1]
[1]Department of Computer Science, University of Texas San Antonio
[2]Department of Computer Science, University of Colorado Boulder
Email: foyzul.hassan@my.utsa.edu, {shaikh.mostfa, xiaoyin.wang}@utsa.edu, edmund.lam@colorado.edu

***Abstract*—Despite the advancement in software build tools such as Maven and Gradle, human involvement is still often required in software building. To enable large-scale advanced program analysis and data mining of software artifacts, software engineering researchers need to have a large corpus of built software, so automatic software building becomes essential to improve research productivity. In this paper, we present a feasibility study on automatic software building. Particularly, we first put state-of-the-art build automation tools (Ant, Maven and Gradle) to the test by automatically executing their respective default build commands on top 200 Java projects from GitHub. Next, we focus on the 86 projects that failed this initial automated build attempt, manually examining and determining correct build sequences to build each of these projects. We present a detailed build failure taxonomy from these build results and show that at least 57% build failures can be automatically resolved.**

I. Introduction

Over the past decade, open software repositories such as GitHub [17], Sourceforge [9] and Google Code [3] are gaining popularity: the number of publicly available repositories have increased tremendously, as software developers are starting to exploit the power of communal open-source development, from small-scale pet projects, to large-scale industrial strength applications (e.g., Android). The availability of open software repositories have presented the software engineering and research communities with a unique opportunity: we now have access to a large corpus of source code from a wide range of software applications that collectively contain immensely rich data. Even at the time of writing this paper, a large number of software engineering techniques have been developed to analyze and mine data (e.g., code, version history, bug reports) from public software repositories [21] [34] [42]. While meta-data of such projects are important, the most significant amount of data is hidden in the syntax and semantics of the code base. Hence, it is often necessary to apply program analysis [39] [43] techniques to these repositories. To do so at a large-scale, it is important that we develop techniques to automate every step of the analysis pipeline, reducing or even eliminating the need for human intervention.

Automating this analysis pipeline is very challenging in general. Best practices dictate that open software repositories store and maintain only source code of software projects, with consensus that outputs (e.g., bytecode and binaries) are expected to built in the local development environments. While some program analysis techniques (e.g., PPA [18]) can be applied to just the source code of the repositories, many useful analysis techniques (e.g., points-to analysis [28], call-graph generation [30], string analysis [29]) depend on either the resolution of dependencies, types, and bindings to extract dependable information, or the build artifacts (byte-code, program traces) as input of the analysis. Furthermore, many well-maintained frameworks (e.g., Soot [25], Wala [6], and LLVM [27]) require built software or a successful build process, and a large number of program analysis techniques are based on these frameworks. Thus, automatic building of project repositories is without doubt, a crucial part of this analysis pipeline.

Yet, in spite of the availability of powerful build automation and integration tools like Ant, Maven and Gradle, the task of building public software repositories is often not entirely an automated endeavor in practice. In this paper, we perform a feasibility study on automatically building software projects downloaded from open software repositories. Specifically, we downloaded the most popular 200 Java projects (by number of stars) from GitHub [17], and applied to each project the default execution command of three popular state-of-the-art Java building tools: Maven [33], Ant [40] and Gradle [24]. We chose GitHub as the source of projects because it is most popular hosting website for open source projects, and it provides a standard interface for meta-data and source code downloading. At GitHub, project users can give stars to a project to show their endorsement, and the number of stars, to a large extent, reflect the size of a project's user group and its popularity among users.

For each project with build failures, we manually examined and uncovered the root causes of the build failures, and manually implemented the fixes to the build scripts where possible. Following this, we categorized all the root causes of failures, and studied whether and how each failure category can be automatically identified and fixed. Our study focuses on Java projects, particularly because it is a mature and widely adopted platform-independent programming language. We also focus on extracting builds from build configuration tools: Ant, Maven and Gradle as they are the three most popular build tools and cover most of open-source Java projects as identified in TravisTorrent [16] data set.

We wish also to emphasize that since detailed build instructions are not always available in open source projects (as shown in our study result in Table II), most users of these

top Java projects and other similar projects in GitHub will typically resort to using default build commands, as we did, in the attempt to build these popular projects. Hence, it is very likely that they will face exactly the same build failures that we have curated here. Therefore, our study is not only useful for automatic software building tools and software engineering researches, but also helpful for project developers and build-tool designers to refine their work and reduce build failures faced by software users.

From our study, we have the following five major findings.

- Gradle and Maven are the two dominating building tools for top Java projects. They are used in 174 of top 200 Java projects.
- 99 of 200 top Java projects cannot be built successfully with default build commands, among which 2 do not have source code, 11 are not using Maven, Ant or Gradle as their building tools, and the remaining 86 projects have various build failures.
- In our hierarchical taxonomy on the root causes of 91 detected build failures (from 86 projects), the leading categories are backward-incompatibility of JDK and building tools, non-default parameters in build commands, and project defects in code / configuration files, which accounts of 19, 33, and 14 build failures, respectively.
- Among 91 build failures, 12 have information in the project's readme file that guides their resolution, and 27 have information in the build failure log that guides the identification of their root causes.
- Among 91 build failures, at least 52 can be automatically resolved by extracting/predicting correct build commands from readme files, exhaustive trial of JDK/build-tool versions, and generation of dummy files.

II. Study Design

A. Research Questions

In our feasibility study, we expect to answer the following research questions.

- **RQ1:** What proportion of top Java projects can be successfully built with default build commands of popular build tools?
- **RQ2:** What are the major root causes of the observed build failures?
- **RQ3:** How easily can root causes of build failures be identified from readme files and build failure logs?
- **RQ4:** What proportion of build failures can be (or have the potential to be) automatically resolved?

B. Study Setup

To perform our study, we downloaded top 200 Java projects from GitHub ranked by the number of stars. We used popularity as project selection criteria since popular projects are more likely to be selected for large-scale software analysis and studies. The downloading was performed as cloning the latest commit as of Aug 30, 2016. Building of software projects not only depends on build commands, but also depends on the build environment (e.g., Java Compiler, build tools). Our build environment includes Ubuntu 14.10, Java SDK 1.8.0_65, Android SDK 24.4.1, Maven 3.3.9, Gradle 3.1, and Ant 1.9.3. We also set environment variables for Java, Android, Ant, Maven and Gradle runtime environments according to their installation guides. When investigating the build failures, we made the necessary customization to the build environment (e.g., reverting to required version of JDK or Android SDK) in order to resolve the respective build issues and achieve a successful build.

III. Study on Successfulness of Default Build Commands (**RQ1**)

To answer **RQ1**, we developed a systematic way to automatically apply the default build commands to each project and determine the outcome of the build attempt. To determine which default build command to use for each specific build tool, our study on build instructions in readme files [8] derived the most frequently used commands, and hence offering the most likelihood of success: for Maven (`mvn compile`), Ant (`ant build`), and Gradle (`./gradlew` and `gradle build` for projects with and without wrappers respectively). A straightforward way of applying build commands is to run them in the root folder. However, we found that among 200 Java projects for study only 35 projects contain build configuration file directly in the root directory. Therefore, we use the following systematic strategy to determine in which folder we apply the default build commands. We first identify a set of folders F that directly contain a build configuration file (i.e., `pom.xml` for Maven, `build.xml` for Ant, `build.gradle` or `gradlew.bat` for Gradle.). If a folder f in F is not directly (or transitively) contained by any other folders in F, we choose f as the folder to apply build commands. If there are multiple folders satisfying this condition, which indicates that multiple sub-projects may be built independently, we choose the folder that transitively contains most Java source files to apply build commands. In case multiple build files present in the folder, we use the file corresponding to the newer building system, as the other files are more likely to be legacy files for maintenance and in many cases older build systems are outdated to build the project. Specifically, Gradle has priority over Maven and Maven has priority oven Ant. Finally, we determine a build to be successful if both two conditions are hold: 1) the exit code of the build process is 0 and 2) no build failure messages are in the build log .

The results of applying default build commands to top 200 projects are presented in Table I. From the table, we have the following observations. First, Maven and Gradle are the dominating build tools used in top Java projects from GitHub. They collectively encompass more than 85% of the projects, hence for automatic software building tools, it is reasonable to focus on only these popular tools. This is also the reason we focus on the build failures of popular build tools in our study. Second, 86 (46%) of 187 projects using Maven, Ant and Gradle build tools failed in the building process. This low successful-build rate essentially shows the necessity of more advanced features for state-of-the-art build automation

TABLE I
OVERALL RESULT OF EXECUTING DEFAULT BUILD COMMANDS

Build Tool	Maven	Gradle	Ant	Other	No Source Code	All
# Projects Built Successful	40	53	8	N/A	N/A	101
# Projects Failed	25	56	5	N/A	N/A	86
# Total Projects	65	109	13	11	2	200

tools. Interestingly, since the top projects are typically well maintained, we can naturally expect build rates to decline in a sample of more average projects in GitHub. Third, Maven has a higher build successful rate than Gradle. While they share similar design ideas, we suspect that Gradle fails in more projects because it allows more customization, and is widely used in Android projects which are more complicated than normal Java projects due to configuration description for Android SDK, NDK and device information. Figure 2 partly verified our guess. Finally, we found 13 projects in our top 200 samples containing no Ant, Maven or Gradle build scripts. Of these, 2 projects have no source code, suggesting that some form of filtering is necessary when processing projects from open software repositories. The other 11 projects contain customized build scripts written by the developers. While it is still possible to analyze and automatically build these project, the lack of standardization and variety of building mechanisms are likely to pose significant challenges.

IV. A TAXONOMY OF ROOT CAUSES OF BUILD FAILURES (**RQ2**)

To answer RQ2, we categorized all the build failures from the 86 projects that failed to be built with default build commands. To clearly confirm the root cause of the build failures, we manually examined and resolved the respective build issues, until we successfully built the project. During this iterative process, we discovered 5 more build failures that were not identified during the builds with default build commands. They are reported after we fixed the first-seen build failures. The reason is that, build tools (actually also compilers) typically stop when they face a fatal error. To make sure we are not biased to the first-seen build failures (which are typically in the earlier stage of building process), we include these 5 build failures into our categorization study.

Our taxonomy generated by the agreement of the first author and the fourth author on build failures by their root causes are presented in Figure 1. In the figure, each colored block represents a category (or sub-category), and the blocks' widths indicate the number of build failures in the category. Finally, higher level categories are linked to their direct sub-categories with elbow lines. We classify the build failures to 3 general categories: environment issues, process issues, and project issues. We will detail these categories and their subcategories in the following subsections.

A. Environment Issues

Environment issues are build failures caused by the change of building environment. They happen when a necessary component in the local system or a remote server becomes unavailable. 31 of the 91 build failures we found are environment issues, and they fall into 3 sub-categories: platform version issues, removed dependency, and requirement of external tools.

Platform Version Issues: This is the largest sub-category of environment issues with 19 of 31 issues fall into it. A platform version issue happens when the successful building of the project relies on a specific historical version of build tools or SDK. Interestingly, though many of these projects exhibited recent code changes, their respective developers have chosen to use older SDK or Build Tools. The reason can be either some dependencies on a specific SDK or build tool version, or developers did not feel that the upgrade was needed (or worth his/her time). An example of platform version issue is shown as BuildLogExample 1. In this example and all following examples, we put the project name (repository name / project name) and commit number at the right corner of the example title. In this example, elasticSearch uses a Gradle feature that became unavailable after version 2.14, so the project can only be built with Gradle 2.13.

BuildLogExample 1 Platform Version Issue 1 *(elasticSearch/elasticSearch: 42a7a55)*

```
A problem occurred evaluating root project '
    buildSrc'.
      > Gradle 2.13 is required to build
          elasticsearch
```

Actually, Build failure logs do not always reveal the root cause of the failures. For example, Example 2 is another build failure example from Facebook Rebound, which shows an error when running Java in the `rebound-android-example:preDexDebug` task of gradle script, but it is very difficult to tell that this error is due to a backward incompatibility of Java 8.

We further studied which build tool or SDK are causing platform version issues. Among the 19 environment issues, 9 requires Java 7 SDK, so we need to downgrade Java from 8 to 7 to solve them; 6 require older Maven versions; 2 require older Gradle versions; and 2 require older Android SDK versions.

BuildLogExample 2 Platform Version Issues 2 *(facebook/rebound: 5017fc9)*

```
* What went wrong:
Execution fail for task ':rebound-android-
    example:preDexDebug'.
> com.android.ide.common.process.
    ProcessException: org.gradle.process.
    internal.ExecException: Process 'command
    '/usr/local/java/jdk1.8.0_65/bin/java''
    finished with non-zero exit value 1
```

Removed Dependency: In Maven and Gradle, dependency Jar files are stored at central repository or developer-specified dependency repositories. Central repositories manage archive

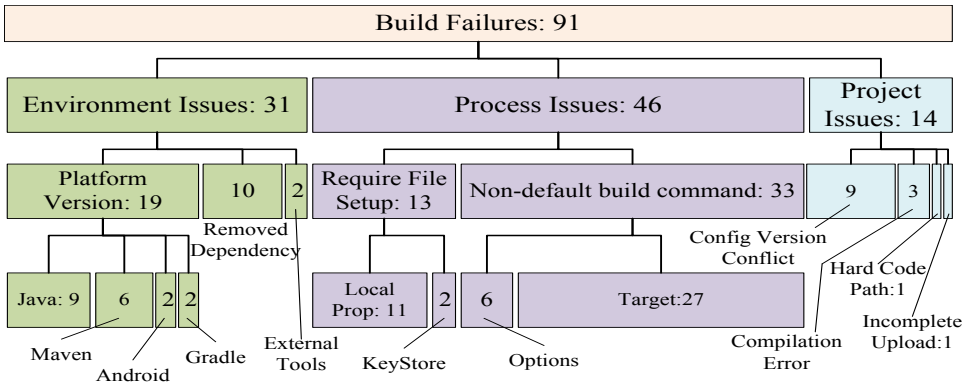

Fig. 1. Build Failure Hierarchy

files to store old versions of Jar files, but in many cases, such archives can be removed from the repository. This removal will unfortunately break the build scripts of any projects relying on them. 9 out of total 10 issues on the removal of dependencies are caused by a certain Jar file being removed from the maven central repository. In Example 3, `com.android.tools.build/gradle 2.0.0-alpha1` no longer exist in the central repository of Maven (perhaps been replaced with a more stable version because the version 2.0.0 does exist).

BuildLogExample 3 Removal of Dependency *(Yalantis/Phoenix: 188f2ec)*

```
> Could not find com.android.tools.build:
    gradle:2.0.0-alpha1.
Searched in the following locations:
https://repo1.maven.org/maven2/com/android/
    tools/build/gradle/2.0.0-alpha1/gradle
    -2.0.0-alpha1.pom
https://repo1.maven.org/maven2/com/android/
    tools/build/gradle/2.0.0-alpha1/gradle
    -2.0.0-alpha1.jar
```

Actually, this type of build failures mostly happens on software projects that are inactive for some time. Among the 10 projects failed with removal of dependency, 7 projects have been inactive for at least 2 years, and 2 have been inactive for at least 6 months by the time we clone the code. The remaining 1 (ACRA/acra) was active, and other versions of the project do build, so we suspect the build failure may be due to a short-term mismatch between configuration file and server status, which gets fixed soon. Although the 9 projects are no longer active, they are still popular among users and users have to find the missing dependency files on the network to build these projects.

External Tools: Two projects require external tools in their build process. These external tools typically facilitate some build steps, such as creating packages in multiple formats. So, build failure happens because external tools are not in the system. Example 4 shows a project calling `fpm` command during the building process, and the `fpm` tool is not available.

BuildLogExample 4 External Tools *(gocd/gocd: a3f77f9)*

```
Execution failed for task ':installers:
    agentPackageDeb'.
> A problem occurred starting process 'command
    'fpm''
```

B. Process Issues

Process Issues are build failures caused by the requirement of additional steps in the building process. To build these projects, we need to either run a command different from the default build command, or some additional parameters, command executions, or settings are required. 46 of the 91 build failures we found are process issues, and they fall into 2 sub-categories as follows.

Non-default Build Command: This sub-category accounts for 33 of the 46 process issues. These build failures happen because the default build command is not the correct build command required to build the project. In Example 5, we get `NoClassDefFoundError`. The project is successfully built if we enter proper the build command `mvn clean install -P 'guice'`, which activates the building profile for `guice`.

BuildLogExample 5 Non-default build command *(roboguice/roboguice: d96250c)*

```
Exception in thread "pool-1-thread-1" java.
    lang.NoClassDefFoundError: org/eclipse/
    aether/spi/connector/Transfer$State
at org.eclipse.aether.connector.wagon.
    WagonRepositoryConnector$GetTask.run(
    WagonRepositoryConnector.java:608)
```

BuildLogExample 6 Require File Setup *(google/iosched: 2531cbd)*

```
A problem was found with the configuration of
    task ':android:packageDebug'.
> File '/home/~/google_iosched/android/debug.
    keystore' specified for property '
    signingConfig.storeFile' does not exist.
```

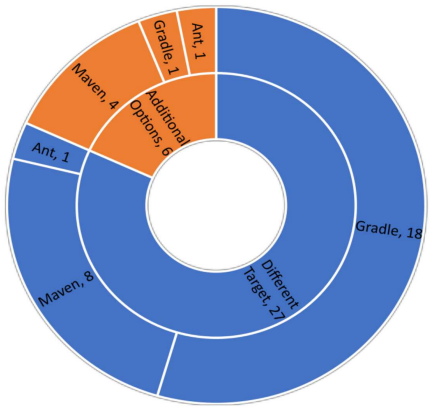

Fig. 2. Non-Default Build Commands Distribution

All 3 build tools we consider defines a special type of parameter that specify the type and phase of building to perform (e.g., whether just clean the project, whether perform unit testing, whether build a release or debug version of an Android apk). These parameters are called "Targets" in Ant, "Lifecycle Commands" in Maven, and "Tasks" in Gradle. In the rest of the paper, we refer to such parameters as "targets", and other parameters as "options". Since the default build command (with default target) does not work, the actual command required can be either a command with a different target or with additional options. In our study, we found 27 build failures for the former case, and 6 build failures for the latter case. The distribution of these build failures in different build tools are presented in Figure 2. The figure shows that Gradle has more failures due to different targets. The reason may be that customized targets [4] (called tasks in Gradle) are used more widely in Gradle.

Require File Setup: This sub-category of 13 build failures are due to the requirement of user generated files during the build process. The two types of required files we found are local property files, in which the user should configure some options or properties such as path to SDK home, and Android keystore files, which the user should generate and sign. Example 6 shows a build failure due to the requirement of a keystore file. It should be noted that, although the user is expected to generate and sign a key with Java keytool, simply copying the default debugging key from Android SDK does not affect the correct building and execution of the project.

C. Project Issues

Project issues are build failures caused by defects in the project itself. These defects can either be code defects that prevent the software from being compiled anyway, or generalization defects that prevent the software from being built on another machine. There are 14 build failures falling into this category, and we classify them into 4 sub-categories: version conflicts in configuration files, compilation errors, hard-coded paths, and incomplete upload.

BuildLogExample 7 Version Conflicts in Configuration Files *(google/iosched: 2531cbd)*

```
> Failed to apply plugin [id 'com.android.
    application']
> Gradle version 2.2 is required. Current
    version is 2.1. If using the gradle
    wrapper, try editing the distributionUrl
    in ~/gradle/wrapper/gradle-wrapper.
    properties to gradle-2.2-all.zip
```

Version Conflicts in Configuration Files: This sub-category contains 9 build failures which are caused by conflicts in the version properties in build configuration files, such as wrong gradle version defined in the wrapper file or version conflicts between parent and child `pom.xml` files in Maven. It should be noted that, since the files with correct versions already exist on the developer's machine, this particular build failure might not manifest in the developer's machine. Example 7 shows a build failure where wrong gradle version (2.1) is specified in the wrapper file. Since developers may already have gradle 2.2 installed on their machines, this defect is not observed on their machines.

Compilation Errors: The 3 build failures in this category happen due to project compilation error. The revision we fetched may be in the middle of global changes as some earlier version builds. Example 8 shows such a compilation error.

BuildLogExample 8 Compilation Failure *(daimajia/AndroidSwipeLayout: d7a5759)*

```
> Compilation failed; see the compiler error
    output for details.
.../library/src/main/java/com/daimajia/swipe/
    SwipeLayout.java:1327: error: illegal
    start of expression
  float willOpenPercent = (
     isCloseBeforeDragged ? ...
```

Hard-Coded Path: Hard-coded path is a common mistake preventing transplantation but it is not seen much in top projects. We do find a hard-coded path error in one project. The path is mentioned in project's configuration file and the build failure is shown in Example 9.

Incomplete Upload: For one project (Example IV-C), we found that one of Android resource file is missing. We suspect that the developers may forget to add the resource file into the software repository for this revision (found in other revisions).

V. Identifying Root Causes of Build Failures (RQ3)

To answer question **RQ3**, we measure for how many build failures the root causes are explicitly mentioned in the build failure logs. For build-failure identification, we used regular-expression based parser and for failure analysis we performed manual analysis on build failures. We also check whether there are build instructions in the readme files / Wiki pages

BuildLogExample 9 Hard-Coded Path *(singwhatiwanna/dynamic-load-apk: d262449)*

```
A problem occurred configuring project ':doi-
    common'.
> The SDK directory '/home/~/153-
    singwhatiwanna_dynamic-load-apk/
    DynamicLoadApk/D:\adt-bundle-windows-
    x86_64-20130219\sdk' does not exist.
```

BuildLogExample 10 Incomplete Upload *(android/platform_frameworks_base: e011bf8)*

```
Execution failed for task ':
    processDebugResources'.
> com.android.ide.common.process.
    ProcessException: org.gradle.process.
    internal.ExecException: Process 'command
    '/home/~/android-sdk-linux/build-tools
    /21.1.2/aapt'' finished with non-zero exit
     value 1
```

of the project [22], and whether following the instructions will avoid the build failure. Specifically, we check whether a keyword (e.g. a platform version for platform-version issues) or a special object (e.g., version number) is mentioned in the build logs or readme files / Wiki pages. For example, Example 1 is considered as being revealed in the build log, while Example 2 is not.

Table II shows the proportion of build failures whose root causes can be identified from readme files, and build logs, respectively. In the table, Line 4 shows the size of the union of build failures in Line 2 and 3, and Line 5 shows the total number of build failures in the category for reference. Columns 2-10 presents the results for each category of build failures. Here we use the level 3 categories as they better reflect the property of the root cause, while level 4 categories are finer-grained and specific to build-tools or file types. Due to space limit, we use some abbreviations in the column names, but the categories in the table are in the exact same order as they are shown in Figure 1.

From the table, we made the following observations: First, root causes of build failures are generally difficult to find. Among 91 build failures, only 39 have their root cause or solution (e.g., the correct build command for projects using non-default commands) mentioned in the project readme files or build failure logs. Second, for several build-failure categories, it is easier to find the root cause or solutions from build log or readme files. Specifically, 11 of 13 build failures in the Require-File-Setup category have the root cause explicitly shown in the build log. The reason is that, when the required file does not exist, the build tools will always report file not found error and report the path where the required file should be placed. Also, for 10 of 33 build failures in Non-Default-Command category, the correct build command is mentioned in readme files. Although the proportion is not high, it shows the possibility of extracting the correct build command from readme files.

It should be noted that, identification of the root cause of a build failure can largely benefit the automatic software building process. However, automatic software building is still possible without knowing the root cause of the build failure. Due to the limited types and concentrated distribution of build failures as shown in Figure 1, it is always possible to try solutions for different root causes until build success or solutions / resources have been exhausted. For example, the automatic building tool may always try different build command targets, and recent versions of built tools / platforms to resolve build failures. Also, the vague relations between build logs and root causes, although hard to understand, may be caught by data mining techniques. Recommended root causes from build logs can save much time for automatic building tools by reducing solutions.

VI. Automatic Resolution of Build Failures (**RQ4**)

To answer question **RQ4**, for each category of build failures, we study whether they can be automatically resolved with heuristics. Specifically, we performed 3 studies to show the feasibility of automatic resolution for 3 major categories of build failures: Non-Default Command, Platform Version Issues, and Require File Setup. For the 65 build failures from these 3 categories, we are able to build 53 of them automatically which are cross validated with manual build.

A. Build Command Extraction and Prediction

To resolve the build failures caused by non-default build commands, we need to find the correct build command to execute. Table II shows that about 1/3 of projects have their correct build command in readme files / Wiki pages, which leads us to check the possibility of extracting commands directly from them.

Named Entity Recognition (NER) [15] is a well-known task to identify a specific type of entities such as people's name, location from natural language texts. It is supported by several popular NLP tool sets such as OpenNLP [14] and Stanford NLP [31]. Build commands in readme files / Wiki pages can also be viewed as a type of entities, and in a previous work [22], we proposed a technique to extract build commands from readme files and Wiki pages, and constructed a training set of readme files / Wiki pages with labeled build commands from 857 of top 1,500 GitHub Java projects which contains such build commands in readme files / Wiki pages. In our study, we apply this technique to the studied 200 projects. Note that, to avoid bias, we excluded all the studied 200 projects from the original training set.

The result of applying NER is presented in Table III. The table shows that we are able to automatically build 5 projects whose correct build command is in readme files / Wiki pages, which is half of the projects having correct build command available. Example 11 shows an example of resolved build failure and the readme files containing the correct build command. Our NER based command extraction

TABLE II
ROOT CAUSE REVEALED

Category	All	Ver. Issue	Depend. Removal	Ex. Tools	Require File Setup	Non-Default Command	Version Conflict	Comp. Error	Hard Coded Path	Incomp. Upload
Readme	12	1	0	1	0	10	0	0	0	0
Build Log	27	1	5	1	11	1	4	2	1	1
Either	39	2	5	2	11	11	4	2	1	1
All	91	19	10	2	13	33	9	3	1	1

tool can extract proper command "mvn clean install" and build the project successfully.

For the projects whose correct build commands are not in the readme file / Wiki pages, we found that most of them use a non-default target but do not need additional options. For Ant, Maven and Gradle if we executed command "ant", "mvn" and "gradle –tasks" respectively, we can obtain the list of all available build targets in the project. But we still need to choose the correct target to use if we do not want to try them exhaustively. Based on our large training set from 857 projects with manually labeled build commands, we are able to find which target is more like the correct building target by calculating the similarity between the target name and all the extracted commands in our training set under the same build tool. Note that this technique is very simple and just taking into consideration the target name, but with the top target we fetch, we are able to successfully build 21 of 24 projects that fails due to using non-default target and not solved with NER. The detailed result is also shown in Table III.

BuildLogExample 11 Failure resolved with NER *(apache/storm: 3a5ecf5)*

```
Readme File:
The following commands must be run from the
    top-level directory.
mvn clean install
If you wish to skip the unit tests you can do
    this by adding -DskipTests to the command
    line.
================================================

Build Log:
[ERROR] Failed to parse plugin descriptor for
    org.apache.storm:storm-maven-plugins
    :2.0.0-SNAPSHOT (~/storm-buildtools/storm-
    maven-plugins/target/classes): No plugin
    descriptor found at META-INF/maven/plugin.
    xml -> [Help 1]
```

Example 12 shows an example of build failure resolved with target estimation. In the list of available non-default targets, we selected "assembleDebug" which has highest ranking. It builds the project successfully because it does not look for API Key, which is required by "build" target and fails the build.

B. Version Reverting

Executing build command with parameter estimation in many cases failed due to incompatible SDK and build tools(e.g Maven, Gradle) versions. To handle such issues, a straightforward way is to revert the versions of SDK and build tools. To study the cost of doing so, we implemented a tool to automatically perform the version reverting, and try to find out how many versions we need to try before finding the correct version. Table IV presents the result of this study, in which rows 2-5 shows the number of projects whose build failures are resolved within a certain type of version reverting, and rows 6-7 shows the average and maximal reverted versions for resolving build failures. The result shows that all 19 failures can be resolved within 10 version reverting, and the average reverting required is 3.8.

However, since it is not always easy to find out which build tool or SDK has version issues, so the automatic building tool may need to try them one by one, and a sum of Java, Maven, and Android trials will bring the worst case to 15 trials.

BuildLogExample 12 Failure resolved with Estimation *(HannahMitt/HomeMirror: 71c860)*

```
ERROR - Crashlytics Developer Tools error.
java.lang.IllegalArgumentException:
    Crashlytics found an invalid API key: null
    .
Check the Crashlytics plugin to make sure that
     the application has been added
    successfully!
Contact support@fabric.io for assistance.
at com.crashlytics.tools.android.
    DeveloperTools.processApiKey(
    DeveloperTools.java:375)
```

C. Dummy File Generation

For build failures in the category of Require-Data-Setup, it is easy to find their root cause (typically shown in the build failure log). Therefore, we can always try to generate a dummy local file as a place holder. Also, in many projects, a sample local file (e.g., local.property.example) is provided, and users can refer to it for what to be put into the local file. In our study, we find that, simply generating an empty local file will resolve 7 of the 13 build failures, and renaming the sample local file (the file whose name is closest to the required file) back will resolve 1 additional build failures.

D. Other Types of Failures

The other types of failures may not have general and straightforward resolution. Removed Dependency and Config Version Conflict are two other large sub-categories with 19 build failures in total. Removed Dependency failures can be

TABLE III
RESOLVED BUILD FAILURES WITH COMMAND EXTRACTION AND PREDICTION

Build Tool / Sub-Type	Maven	Gradle	Ant	All	Target	Para
NER	2	3	0	5	4	1
Target Estimation	5	15	1	21	21	0
All fixed	7	18	1	26	25	1
Not fixed	5	1	1	7	3	4

TABLE IV
RESOLVED BUILD FAILURES WITH VERSION REVERTING

Platform	Java	Maven	Gradle	Android	All
Revert 1 version	9	0	0	0	9
Revert 2-5 versions	0	0	1	2	3
Revert 6-10 versions	0	6	1	0	7
Revert 11+ versions	0	0	0	0	0
Max # reverted versions	1	10	9	4	10
Avg # reverted versions	1	7.3	6.5	3	3.8

easily resolved if the dependency file can be found in large Jar repositories (e.g., Maven Central, Java2S) or Google, but the difficulty varies for each Jar. A potential solution is to search for references to the Jar file in other projects' configuration files, and try to fetch the Jar file from their project folder or referred server.

A large portion of Config Version Conflict failures (7 of 9) are due to out-of-date Gradle wrappers. These failures are easy to find and resolve, because we just need to update the version specified in gradle wrapper to the same as the gradle script. However, the other 2 failures are due to mismatches between parent and child pom files in maven. To resolve them, we need to perform in-depth analysis of pom files and their dependencies.

VII. DISCUSSIONS

A. Threats to Validity

The major threat to the internal validity of our evaluation is the correctness of manual process in our experiment, including the implementation of our approach, the labeling of build commands, and the manual build process. To reduce this threat, we carefully performed all these process, and double checked the consistency throughout the data sets. The major threat to the external validity of our evaluation is that our findings may be only applied to our subject data set. To reduce this threat, we used a large number of top Java projects for evaluating project building. Our experiment confirms that these projects cover various build configuration systems.

B. Lessons Learned for Automatic Software Building

It is a necessity. Our study finds that, about half of the top Java projects cannot be straightforwardly built with default build commands. This is a very disconcerting fact: consider having to create a data set of 200 built Java projects (not very large for mining or training purposes), while a large number can be automatically built with simple default build commands, a researcher still needs to manually build 100 projects.

Furthermore, referring to our experiment on top 200 projects again, among the 86 projects that have build failures, 59 do not contain sufficient documentations (e.g., in readme files) that describe the correct building instructions or common building pitfalls. A developer wanting to build such projects have little choice but to engage in a "trial by error" attempt to manually iterate through known build targets. This is obviously a time consuming process and in no way a scalable process, if done manually. It is a clear challenge that has to be addressed if software engineering researchers aspire to develop large-scale program analysis techniques that require massively large sets of build artifacts. Perhaps one can consider simply to rely on a best effort automated use of default build commands to build half of the projects in a large corpus. This naive solution however, might introduce unknown bias into downstream research processes. In essence, this 'build' problem is not simply just a matter of productivity, but likely a hurdle of the software engineering research methodologies.

It is feasible. Our study shows that, among the 86 projects with build failures, 26 projects can be built successfully with relatively simple heuristics. For instance, reverting platform to all possible versions, and generating a dummy property file. Furthermore, 26 additional projects can be built successfully by the NER technique and mining commands from a large set of readme files. The 7 projects with unmatched gradle wrappers and missing dependency Jar files also have the potential to be automatically built by fixing the mismatches between gradle wrapper and gradle scripts, and searching and downloading the missing Jar files from other servers (e.g., Java2S [7]). These numbers show that, more than 75% of the projects with build failures can be automatically resolved with simple rules or some advanced techniques, so a tool combining default command execution and the resolution techniques above may achieve an overall build successful rate of 80%. A tool that systematically binds together these heuristics to fix build scripts and strategies to iteratively try build alternatives, would not only reduce the manually effort of software engineering researchers, as well as enlarge the amount of build data that they can extract from open repositories.

The challenges. Our study has also identified several build-failure categories whose automatic resolution can be difficult. For instance, when the building process requires external tools, when a dummy local property file cannot enable successful build, or when complicated build commands are required but no instructions are available in the readme files / Wiki pages. To handle these cases, a tool needs to automatically analyze the project files structures, build configuration files, and also mine and analyze discussion threads from online forums such as StackOverflow [10]. These research tasks not only can be challenging but also of great value. Note that bringing the overall build successful rate from 80% to 90% will reduce half of the manual build effort required by researchers.

Another challenge for automatic building tools is the time and resource consumption of building each project. A successful build is possibly preceded by a large number of unsuccessful attempts. For example, the building tool may try a lot of build-tool versions to find the version that is compatible with the project. Although the build process is automated,

considering the large number of projects in open repositories, the performance can still be a major concern. Hence, it is very important that the techniques that we use must be easily scalable (i.e., large volumes of input can be handled by adding more compute strength to the cluster). Also these techniques must be highly optimized, for instance, in the manner they extract/use information from build failure logs and possibly learn from previous attempts (e.g., recognize and prune away known hopeless attempts).

C. Lessons Learned for Build-Tool Developers

Our study has shown that the backward incompatibility of build tools is one of the leading root causes of build failures. As time goes by, a project will no longer be buildable with the newest version of built tools. While backward incompatibility is often unavoidable, build tool developers should maintain and make available all previous versions of the tool, and force the build configuration file to specify the version that the project is built on. In fact, Gradle Wrapper [5] is a built-in mechanism in Gradle that automatically downloads and uses the proper gradle version. Interestingly, we found out-of-date gradle wrapper files in 7 projects, and the presence of such fault in a project often fatally derails the building process. We believe that these consistencies between the wrapper and the build script should be enforced by the build tool itself. Furthermore, a lot of projects are using non-default build commands and parameters without having any instructions in their readme files. An excellent feature that build tool developers should consider supporting is one that records the developers' command sequences when they build the software and automatically generate some wrapping scripts that repeats their build actions.

D. Lessons Learned for Project Developers

The Version Conflicts in Configuration Files build failure category contains 9 build failures that are caused by defects in the software project itself. The 3 code compilation errors are due to incomplete commits (the code revision we downloaded is in the middle of a global change and thus cannot be built). The rest build failures are all caused by defects that prevent the software to be built on another machine, such as hard-coded paths, miss-uploaded files, and out-of-date version in wrapper files or configuration files. This calls for a testing of the project on a different machine when a change is made to configuration files / wrapper files or build dependencies. In fact, this can be forced by some code reviewing tools like Gerrit [1]. And the automatic software building tools can also serve as a testing tool to report build-related defects (such as out-of-date gradle wrappers) to project developers.

E. YML Files and Continuous Integration

For Continuous Integration (CI) [23], some projects adopt Travis CI and include a yml file in the project repository to specify build steps for CI process. In our study, we find 95 of the top 200 projects to include such yml files. Using yml files to build the project requires specific integration configurations and software support from Travis CI, so in our study we do not use yml files for automatic building.

VIII. Related Works

Study on Build Failures. On the study of building errors, Hyunmin et al. [35] carried out an empirical study to categorize build errors at Google. Their study shows that missing types and incompatibility are the most common type of build errors, which are consistent with our findings. However, they also find many semantic or syntactic errors, which are very rare in our study. This is not surprising, since their study focused on the build errors in the original environments, while our study focuses on the build errors due to environment changes. Also, since our projects are committed versions in the repository, with support of current IDEs, they are more likely to be built successfully in original environment, and have fewer code related errors. Recently, Tufano et al. [41] studied the frequency of broken (not compilable) snapshots and likely causes of broken snapshots. Sulír et al. [37] performed build failure analysis based on build log text categorization to find out frequency and reasons for build errors. McIntosh et al. [32]'s study also supports that for modern build systems build maintenance effort on external dependency is higher than than internal dependency management. Compared to these works which analyze only build logs, we performed detailed manual analysis and building to find out and confirm the root causes of the build failures. As an example, dependency issues are found to be a major reason of build failures by previous works, but we found that such dependency issues may be caused by build-plug-in version errors or a wrong build command used. So simply adding the dependency back to the project may not resolve the build failure.

Automatic Software Building. Lämmel et al. performed semi-automatic building of Ant-based Java projects (i.e., the QUAATLAS corpus [19]) in their API statistics study [26] . The thesis version [36] of the work details the software building process, which includes automatic scripts to locate Ant configuration files and to run the Ant command. However, the work uses only predefined Ant commands sets, and does not take advantage of information in readme files, so it is similar to our baseline approach but specific for Ant. On migration of build configuration files, AutoConf [2] 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. Most recently, Gligoric et al. [20] proposed an approach to automate the migration of various building configuration files to CloudMake configuration files based on building execution with system-level instrumentation.

Analysis of Building Configuration Files. Analysis of build configuration file 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, Aoumeur [13] proposed a Petri-net based model to describe the dependencies

in build configuration files. Adams et al. [11] 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. [12] proposed a fault localization approach for make files, and SYMake [38] 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.

IX. Conclusions

Software building takes significant amount of time for software engineering researchers before analyzing projects. Developers may also need to build a list of third party tools before they can use them. This paper comes up with the first study on the build failures found in top Java projects, and whether such failures can be resolved with automatic tools. We have constructed a taxonomy for the root causes of build failures, and study the distribution of build failures in different categories. Specifically, we found 91 build failures in 86 of the 187 Java projects that use Maven, Ant, and Gradle for their building process, and we found the leading root causes of build failures are backward-incompatibility of JDK and building tools, non-default parameters in build commands, and project defects in code / configuration files. Finally, 52 of the build failures can be resolved automatically, and additional 6 build failures have the potential to be resolved automatically.

Acknowledgments This material is based on research sponsored by NSF Award CCF-1464425 and DARPA grant under agreement number FA8750-14-2-0263.

References

[1] Gerrit code review - a quick introduction. https://review.openstack.org/Documentation/intro-quick.html/, accessed: 2016-10-20
[2] Gnu autoconf - creating automatic configuration scripts. http://www.gnu.org/software/autoconf/manual/index.html, accessed: 2015-10-25
[3] Google code âĂŤ project hosting. https://code.google.com/hosting, accessed: 2009-08-06
[4] Gradle task type. https://docs.gradle.org/3.3/dsl/index.html#N1000C/, accessed: 2017-07-04
[5] The gradle wrapper. https://docs.gradle.org/current/userguide/gradle_wrapper.html/, accessed: 2016-10-20
[6] Ibm. the t. j. watson libraries for analysis (wala). http://wala.sourceforge.net, accessed: 2012-03-20
[7] Java2s jar archive. http://www.java2s.com/Code/Jar/CatalogJar.htm, accessed: 2016-09-20
[8] Readme. https://en.wikipedia.org/wiki/README, accessed: 2016-09-20
[9] The sourceforge story. http://web.archive.org/web/20110716044546/http://itmanagement.earthweb.com/cnews/article.php/3705731, accessed: 2012-04-12
[10] Stackoverflow. http://stackoverflow.com/, accessed: 2016-09-25
[11] Adams, B., Tromp, H., De Schutter, K., De Meuter, W.: Design recovery and maintenance of build systems. In: ICSM. pp. 114–123 (2007)
[12] Al-Kofahi, J., Nguyen, H.V., Nguyen, T.N.: Fault localization for build code errors in makefiles. In: ICSE Companion. pp. 600–601 (2014)
[13] Aoumeur, N., Saake, G.: Dynamically evolving concurrent information systems specification and validation: A component-based petri nets proposal. Data Knowl. Eng. 50(2), 117–173 (Aug 2004)
[14] Apache: Opennlp (2010), http://opennlp.apache.org
[15] Balasuriya, D., Ringland, N., Nothman, J., Murphy, T., Curran, J.R.: Named entity recognition in wikipedia. In: Proceedings of the 2009 Workshop on The People's Web Meets NLP: Collaboratively Constructed Semantic Resources. pp. 10–18. Association for Computational Linguistics (2009)
[16] Beller, M., Gousios, G., Zaidman, A.: Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: MSR. pp. 447–450. IEEE Press, Piscataway, NJ, USA (2017), https://doi.org/10.1109/MSR.2017.24
[17] Charles, P.: Project title. https://github.com/charlespwd/project-title (2013)
[18] Dagenais, B., Hendren, L.: Enabling static analysis for partial java programs. In: OOPSLA. pp. 313–328 (2008)
[19] De Roover, C., Lammel, R., Pek, E.: Multi-dimensional exploration of api usage. In: ICPC. pp. 152–161 (2013)
[20] Gligoric, M., Schulte, W., Prasad, C., Van Velzen, D., Narasamdya, I., Livshits, B.: Automated migration of build scripts using dynamic analysis and search-based refactoring. In: ACM SIGPLAN Notices. vol. 49, pp. 599–616. ACM (2014)
[21] Hassan, A.E.: The road ahead for mining software repositories. In: FoSM. pp. 48–57. IEEE (2008)
[22] Hassan, F., Wang, X.: Mining readme files to support automatic building of java projects in software repositories: Poster. In: ICSE Companion. pp. 277–279 (2017)
[23] Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: ASE. pp. 426–437. ASE 2016, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2970276.2970358
[24] Ikkink, H.K.: Gradle Dependency Management. Packt Publishing (2015)
[25] Lam, P., Bodden, E., Hendren, L., Darmstadt, T.U.: The soot framework for java program analysis: a retrospective
[26] Lämmel, R., Pek, E., Starek, J.: Large-scale, AST-based API-usage analysis of open-source Java projects. In: SAC. pp. 1317–1324 (2011)
[27] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: CGO. pp. 75– (2004)
[28] Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: CC. pp. 153–169. Springer (2003)
[29] Li, D., Lyu, Y., Wan, M., Halfond, W.G.: String analysis for java and android applications. In: ESEC/FSE. pp. 661–672. ACM (2015)
[30] Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for java. In: PLAS. pp. 139–160 (2005)
[31] Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S.J., McClosky, D.: The Stanford CoreNLP natural language processing toolkit. In: ACL. pp. 55–60 (2014), http://www.aclweb.org/anthology/P/P14/P14-5010
[32] Mcintosh, S., Nagappan, M., Adams, B., Mockus, A., Hassan, A.E.: A large-scale empirical study of the relationship between build technology and build maintenance. Empirical Softw. Engg. 20(6), 1587–1633 (Dec 2015), http://dx.doi.org/10.1007/s10664-014-9324-x
[33] Miller, F.P., Vandome, A.F., McBrewster, J.: Apache Maven. Alpha Press (2010)
[34] Mostafa, S., Wang, X.: An empirical study on the usage of mocking frameworks in software testing. In: QSIC. pp. 127–132. IEEE (2014)
[35] Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R.: Programmers' build errors: A case study (at Google). In: ICSE. pp. 724–734 (2014)
[36] Starek, J.: A large-scale analysis of Java API usage (2010)
[37] Sulír, M., Porubän, J.: A quantitative study of java software buildability. In: PLATEAU. pp. 17–25. ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/3001878.3001882
[38] Tamrawi, A., Nguyen, H.A., Nguyen, H.V., Nguyen, T.N.: Symake: A build code analysis and refactoring tool for makefiles. In: ASE. pp. 366–369 (2012)
[39] Tang, H., Wang, X., Zhang, L., Xie, B., Zhang, L., Mei, H.: Summary-based context-sensitive data-dependence analysis in presence of callbacks. In: POPL. pp. 83–95 (2015)
[40] Tilly, J., Burke, E.M.: Ant: The Definitive Guide. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edn. (2002)
[41] Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D.: There and back again: Can you compile that snapshot? Journal of Software: Evolution and Process 29(4) (2017)
[42] Wang, X., Zhang, L., Tanofsky, P.: Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In: ISSTA. pp. 199–210. ACM (2015)
[43] Zhang, H., Kuan Tan, H.B., Zhang, L., Lin, X., Wang, X., Zhang, C., Mei, H.: Checking enforcement of integrity constraints in database applications based on code patterns. JSS (2011)