

Detecting Build Dependency Errors in Incremental Builds

Jun Lyu
Nanjing University
Nanjing, China
lvjun@smail.nju.edu.cn

Shanshan Li*
Nanjing University
Nanjing, China
lss@nju.edu.cn

He Zhang
Nanjing University
Nanjing, China
hezhang@nju.edu.cn

Yang Zhang
Nanjing University
Nanjing, China
zhangyang2021@smail.nju.edu.cn

Guoping Rong
Nanjing University
Nanjing, China
ronggp@nju.edu.cn

Manuel Rigger
National University of Singapore
Singapore, Singapore
rigger@nus.edu.sg

ABSTRACT

Incremental and parallel builds performed by build tools such as Make are the heart of modern C/C++ software projects. Their correct and efficient execution depends on build scripts. However, build scripts are prone to errors. The most prevalent errors are missing dependencies (MDs) and redundant dependencies (RDs). The state-of-the-art methods for detecting these errors rely on clean builds (i.e., full builds of a subset of software configurations in a clean environment), which is costly and takes up to a few hours for large-scale projects. To address these challenges, we propose a novel approach called ECHECKER to detect build dependency errors in the context of incremental builds. The core idea of ECHECKER is to automatically update actual build dependencies by inferring them from C/C++ pre-processor directives and Makefile changes from new commits, which avoids clean builds when possible. ECHECKER achieves higher efficiency than the methods that rely on clean builds while maintaining effectiveness. We selected 12 representative projects, with their sizes ranging from small to large, with 240 commits (20 commits for each project), based on which we evaluated the effectiveness and efficiency of ECHECKER. We compared the evaluation results with a state-of-the-art build dependency error detection tool. The evaluation shows that the F-1 score of ECHECKER improved by 0.18 over the state-of-the-art method. ECHECKER increases the build dependency error detection efficiency by an average of 85.14 times (with a median of 16.30 times). The results demonstrate that ECHECKER can support practitioners in detecting build dependency errors efficiently.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

Build Tool, Build System Maintenance, Build Dependency Errors

*Corresponding author

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

ISSTA '24, September 16–20, 2024, Vienna, Austria

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0612-7/24/09

<https://doi.org/10.1145/3650212.3652105>

ACM Reference Format:

Jun Lyu, Shanshan Li, He Zhang, Yang Zhang, Guoping Rong, and Manuel Rigger. 2024. Detecting Build Dependency Errors in Incremental Builds. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, September 16–20, 2024, Vienna, Austria. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3650212.3652105>

1 INTRODUCTION

Common build systems [26], such as GNU MAKE [16], SCONS [42], ANT [4], MAVEN [30] and GRADLE [19], rely on build scripts to perform builds correctly and efficiently. However, developing such build scripts is not an easy task. Although automated script generators such as AUTOTOOLS [5] and CMAKE [29] may help, there is still a need to manually enumerate build dependencies for custom software. For real large-scale projects, dependency enumeration is error-prone. In particular, in C-based projects, 52.68% of the build errors are dependency-related errors [43].

Common build dependency errors for C-based projects are *Missing Dependencies* (MDs) and *Redundant Dependencies* (RDs). The root causes of both dependency errors are the differences between the dependencies declared in the build script (a.k.a., declared build dependencies) and the dependencies used by the build system in the actual build (a.k.a., actual build dependencies). Actual build dependencies refer to the dependencies used by the build system in the build process of the targets [9]. MDs refer to the actual build dependencies that are erroneously missing from the declared build dependencies. GNU MAKE, which is one of the most widely-used build systems [32–34], performs incremental builds based on the dependencies declared in the build script [16]. MDs prevent GNU MAKE from recompiling programs after they have been modified and regenerating all the targets that contain them, resulting in incorrect incremental builds [26]. RDs refer to declared build dependencies which declare dependencies that are not the actual build dependencies of the target. RDs cause the build system to perform unnecessary incremental builds. In addition, RDs enforce the targets that could be executed in parallel to be executed sequentially, reducing the build efficiency [26].

The software engineering (SE) community has proposed many methods to detect build dependency errors [53, 54]. The state-of-the-art methods utilize a clean build to obtain the actual build dependencies [7, 12, 26, 44, 52]. Such methods often detect dependency errors using actual build dependencies together with or without declared build dependencies. Running such a tool for each commit is desirable because each commit can introduce dependency errors

```

1. SRC = $(wildcard src/*.c)
2. DEPS = $(wildcard deps/*/*.c)
3. OBJS = $(DEPS:.c=.o)
4. $(BINS): $(SRC) $(OBJS)
   $(CC) $(CFLAGS) -o $@ src/$(@:.exe=).c
   $(OBJS) $(LDFLAGS)

```

Figure 1: Dependency errors form *Clib* Makefile

and it is more cost-effective to fix an error as early as possible in the development process. The current state-of-the-art methods require a clean build for each detection, making this approach costly. Clean builds can be very time-consuming for large projects (e.g., *OpenCV* clean builds on our machine in over an hour). It is difficult for practitioners to quickly perform build dependency error detection with state-of-the-art methods for each commit.

Consider the following example from the *clib* Makefile shown in Figure 1. The build script (Makefile) declares all files with the suffix “.c” as prerequisites (see line 4), but no other files (e.g., header files) are included. Suppose a practitioner changes a header. GNU MAKE will not regenerate any of the files containing the header. This means that the header files are MDs, which may result in incorrect builds. Changes to MDs files that should be rebuilt and relinked in an incremental build may not trigger the correct incremental build as expected. As with other state-of-the-art approaches, Buildfs [44] performs a clean build of the project once per detection. Even if a new commit implements only a minor modification of one of the header files, the current methods still require a clean build to detect dependency errors. The clean build is time-consuming for large-scale projects, rendering the dependency detection methods relying on it non-interactive and expensive.

To address these challenges, we propose an approach called ECHECKER to detect build dependency errors for C-based and GNU MAKE-based projects. The core idea behind ECHECKER is to infer actual build dependency changes based on code changes in commits. In detail, actual build dependencies can be inferred from pre-processor directives and Makefile changes based on new commits, rather than obtaining actual build dependencies by executing an entire clean build. Our key observation is that the actual build dependencies of the target only change when the pre-processor directives [37] of source code (include directive, i.e., `#include <example.h>`) and the build commands in the Makefile are changed. The pre-processor directives and build commands indicate which files the compiler should include in the source file for building, i.e., they indicate the actual build dependencies. However, this is insufficient to detect some dependency errors when dependencies only emerge during building (e.g., a header file might be automatically generated in the build, called hidden dependencies) [26]. Thus, ECHECKER monitors the incremental builds that are triggered from application commits, which can help to further identify actual build dependencies. ECHECKER generates only a small number of false positives if GNU MAKE cannot rebuild a project due to improper build commands. These false positives are relatively rare because they come from improperly compiled commands (i.e., command “In-s” cf. Section 5.2) rather than from the design flaws in ECHECKER.

We evaluate the effectiveness and efficiency of ECHECKER in 12 representative projects with 240 commits (20 commits per project) and compared it with the latest publicly available state-of-the-art method (Buildfs [44]). We simulate continuous code updates (20

commits) in real-world development by selecting consecutive commits. The evaluation results show that ECHECKER is more effective and efficient at detecting build dependency errors than Buildfs. Specifically, ECHECKER detects 1,635 MDs and 663 RDs with only 32 false positives. ECHECKER improves detection efficiency over Buildfs by 85.14 times (with the median at 16.30 times), while significantly reducing the time it takes to detect build dependency errors. ECHECKER shows its great potential to detect build dependency errors efficiently. As the frequency of commits increases, developers can continue to reap the benefits.

The main contributions of this paper are as follows.

- We propose a novel approach called ECHECKER to detect build dependency errors. Unlike existing methods, we improve the detection of build dependency errors using incremental builds and avoid time-consuming clean builds when possible.
- We present the idea of inferring actual build dependencies based on code changes in commits, which is useful for the detection of build dependency errors.
- We evaluate ECHECKER and demonstrate its significant improvement in the efficiency of detecting dependency errors with satisfactory effectiveness. ECHECKER can be expected to facilitate software development by efficiently and continuously detecting build dependency errors.

2 BACKGROUND AND MOTIVATION

This section introduces the basics of build dependency errors and further motivates the issue.

Build dependencies and dependency errors. This build process involves analyzing the build scripts that contain the targets that need to be built or not [16]. Dependencies are required for the build of the target and the build commands to generate the target. In Figure 1, the build target “BINS” requires C source files of the format (“src/*.c”) at build time; thus, the build target “BINS” is said to depend on dependency “src/*.c”. Targets and dependencies form a directed acyclic graph, known as a dependency graph [9]. In this graph, targets are nodes and target dependencies are edges (i.e., “BINS” is a node, “src/*.c” is an edge).

Typically, each build has two kinds of dependencies: declared build dependencies and actual build dependencies [9]. The actual build dependency graph is derived from the declared build dependencies. The Makefile forms declared build dependencies through targets and prerequisites. Actual build dependencies are needed during the build to produce the correct build of targets—thus, they are a dynamic property; for example, if BINS and all its dependencies are up-to-date, then “BINS” will be built correctly. If a dependency is the actual build dependency of the target (“BINS”), the dependency (“src/*.c”) must exist, be constructed, and be updated when the target is built, regardless of whether it is declared as an actual build dependency of the target [9]. GNU MAKE will decide to build a minimal set or a re-build target based on declared build dependencies and modification time. An error will be generated if the actual build dependencies do not match the declared build dependencies.

Motivation. Buildfs [44] is considered the state-of-the-art approach. Buildfs improves detection efficiency and supports the detection of dependency errors in C and Java projects. Buildfs cuts the build task into a series of build targets and obtains the actual build dependencies for each target by performing a clean build on each target. While Buildfs has shown to be highly effective, it has issues in the context of conditional compilation. Conditional compilation refers to the execution of subsequent builds based on the pre-order building or conditions (e.g., environment variables and build parameters). Buildfs ignores the existence of conditional compilation, which can omit or redundantly build artifacts, negatively affecting build efficiency and effectiveness. Since Buildfs builds each target individually, the build system is unable to correctly determine the build conditions. This causes false positives and false negatives to occur during the build process. Furthermore, Buildfs is unable to detect RDs.

Using state-of-the-art approaches like Buildfs, practitioners cannot obtain timely feedback on dependency errors with frequent updates. Typically, they comprehensively test each commit to ensure that it is free from hidden bugs. Thus, ideally, test results would be available immediately after each test. However, in reality, the testing phase is a time-consuming process (e.g., it takes hours to detect build dependency errors) and therefore feedback can significantly lag. As a result, developers need to wait longer to see the results of these tests, slowing down their development, and fixing issues potentially only after having started a new task. To address this challenge, we propose a new approach capable of quickly detecting build dependency errors with improved effectiveness. It can be implemented locally or integrated into Continuous Integration (CI) to help practitioners quickly perform dependency error detection during development.

3 APPROACH

We propose an approach called ECHECKER to detect build dependency errors. The core idea of ECHECKER is to efficiently detect build dependency errors by inferring the actual build dependencies. Our key observation is that actual build dependencies of the target change only when the source pre-processor directives and the build commands in the Makefile are changed. Specifically, ECHECKER infers the actual build dependencies based on changes to pre-processor directives or build commands in the commit. This allows us to obtain new actual build dependencies without performing a clean build. In addition, ECHECKER monitors incremental builds, which in turn improves the effectiveness of detection by allowing our approach to successfully handle hidden dependencies. ECHECKER transforms the traditional clean build-based dependency error detection to incremental build-based, which improves detection efficiency while ensuring effectiveness.

3.1 Overview

Figure 3 illustrates how ECHECKER works. ECHECKER takes the project, commit, and the historical actual build dependency graph of clean builds as inputs. Assuming that the historical actual build dependency graph of clean builds exists, ECHECKER detects build dependency errors in three stages. First, ECHECKER applies changes of a commit to trigger an incremental build and monitors the execution of the incremental builds to obtain the actual build dependency

graph of incremental builds (step ①). Second, ECHECKER analyzes the specifics of the commit. ECHECKER focuses on whether the commit changes the pre-processor directives of source files and the Makefile (steps ②–④). Then, ECHECKER infers the actual build dependency graph by leveraging the results of step ②–step ④ (step ⑤). Finally, ECHECKER detects build dependency errors based on the updated actual build dependency graph (step ⑥). If the historical actual build dependency graph of clean builds does not exist, ECHECKER performs a clean build to detect dependency errors and saves the actual build dependency graph as input for the next detection. Note that, for the detection of multiple commits merely one clean build is required for ECHECKER. Then it will continuously update the actual build dependency graph by inferring actual build dependencies.

3.2 Inferring Actual Build Dependencies

This subsection describes in detail how ECHECKER infers the actual build dependency graph in three stages: monitoring incremental builds execution, pre-processor directives, and Makefile change analysis, and inferring actual build dependencies.

3.2.1 Monitoring Incremental Builds Execution. ECHECKER monitors the execution of the incremental builds to obtain the actual build dependency graph of incremental builds (step ①).

The actual build execution information has to be retrieved by tracing the build process executed by GNU MAKE. In a file system-based build, the build is composed of a series of operations on files, e.g., *reads*, *deletes*, and *creates* [24, 46]. The build execution information is implicitly exposed in the system calls. GNU MAKE creates a sub-process for each build target and executes the corresponding build commands in the sub-process. The execution of the build commands is essentially the process of manipulating the underlying files. Therefore, it is possible to trace the operations performed on the underlying files during the build process to obtain the actual build dependencies for each build target. Based on the type and parameter information of all captured system calls, the files manipulated by each process can be divided into input files (files called during the build process, i.e., build process reads “/src/options.h” in Figure 3) and output files (files newly created or written during the build process, i.e., build process creates “/src/options.o” in Figure 3). By monitoring the build process, ECHECKER can obtain actual build dependencies.

ECHECKER is designed to report potential dependency errors only to the project, not external dependencies, such as standard libraries. ECHECKER excludes external dependencies of the project by file paths. As shown in Figure 2, ECHECKER excludes the files (blue text in the figure) by their file path, not the external dependencies (not in the project file path, e.g., “/Example-master/” in the figure). For

```

/usr/include/x86_64-linux-gnu/bits/iscanonical.h
/usr/include/ctype.h
/Example-master/src/options.h
/Example-master/a.c
/usr/lib/x86_64-linux-gnu/libm-2.31.so
...

```

} Example-master/src/a.o

Figure 2: Tracing Build of the Example Projects

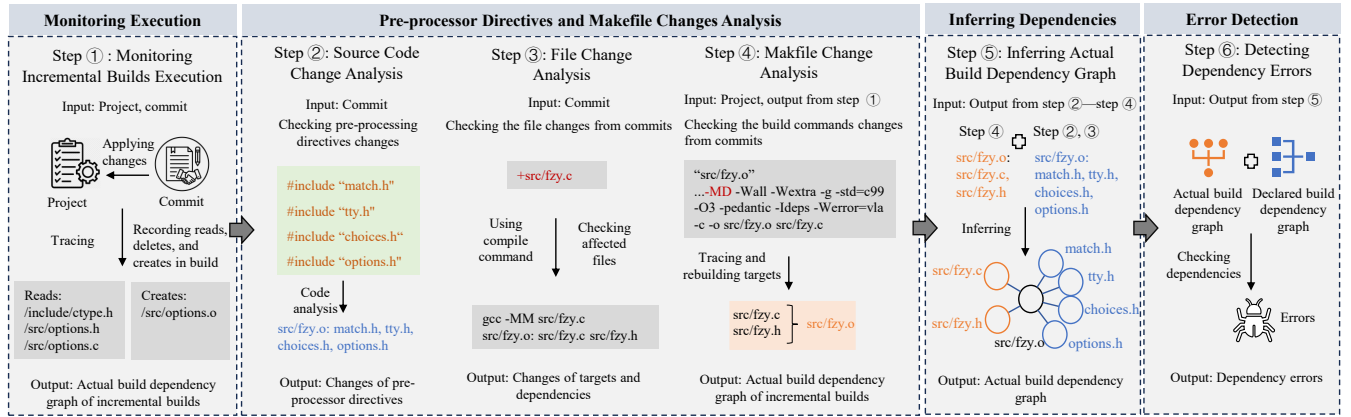


Figure 3: Overview of ECHECKER

example, most projects rely on the standard library (e.g., `/x86_64-linux-gnu/bits/iscanonical.h`) in the figure). ECHECKER may report a large number of missing basic libraries if they are not excluded. In reality, it is difficult for practitioners to include all external libraries needed in a Makefile. Many tools (e.g., AUTOTOOLS [5] and CMAKE[29]) are designed to help practitioners automatically generate a Makefile without having to consider external libraries.

3.2.2 Pre-processor Directives and Makefile Change Analysis. Monitoring incremental builds helps to obtain hidden dependencies. However, the actual build dependency graph for incremental builds is not complete enough, because the execution of incremental builds can be affected by dependency errors that we aim to find. For example, the incremental builds system does not perform re-building of the source files when the header files in Figure 1 are changed. The idea of ECHECKER is that the incomplete actual build dependency graph obtained by monitoring can be complemented by analyzing the pre-processor directives and the Makefile, which is divided into two phases: pre-processor directives change analysis and build commands change analysis (steps ②–④).

Pre-processor directives change analysis. The changes from commits include changes to pre-processor directives and source files.

For pre-processor directives, ECHECKER identifies changes made to the pre-processor directives by a commit (e.g., additions and deletions). All nodes that depend on a file are then updated in the actual build dependency graph, and actual build dependencies are added and removed based on the changes. At build time, GNU MAKE recognizes the files referenced by the source file (files beginning with the `#include`) and includes those files in the build of the source file. Files starting with `#include` are considered to be actual build dependencies. Figure 4 illustrates how ECHECKER updates the actual build dependencies when the pre-processor directives are changed. The pre-processor directives of `/Example-master/a.c` are removed and added respectively (lines 2 and 3). ECHECKER updates each node containing `/Example-master/a.c` in the actual build dependency graph (lines 5–7).

For the files that are added or deleted, ECHECKER checks their file types by the suffix to determine subsequent actions. For source code types (e.g., `.c`, `.cpp`, `.cc`, `.h`, `.hpp`), ECHECKER uses the compiler commands (e.g., `gcc -MM`) [15] to obtain the files on which

the added or deleted files depends and then add or remove these files (nodes) from the actual build dependencies. The files that have a non-source file type are updated during the incremental build monitoring step and do not need to be processed again here. For renamed files, ECHECKER uses the new filename to replace the original actual build dependencies. Figure 5 is an illustration of ECHECKER infers the actual build dependencies when the files are

```

1. //Pre-processor directives changes in Example-master/a.c
2. + #include b.h
3. - #include src/options.h
4. //Inferring Actual build dependencies for Example-master/a.c
5. If a.c in actual build dependency[items]
6. actual build dependency[items].add(b.h)
7. actual build dependency[items].remove(src/options.h)
    
```

Figure 4: Inferring Actual Build Dependencies for Changes to Pre-processor Directives

```

1. //Files Changes in Example-master
2. - src/options.c
3. + b.c
4. + src/Readme
5. //Inferring Actual build dependencies for Example-master
6. gcc-MM src/options.c
7. src/options.o: src/options.c src/options.h
8. gcc-MM b.c
9. b.o: b.c b.h
10. actual build dependency[src/options.o].remove(src/options.c, src/options.h)
11. actual build dependency[b.o].add(b.c, b.h)
    
```

Figure 5: Inferring Actual Build Dependencies for Changes to Files

```

1. //Makefile Changes in Example-master
2. $(OUT)/b.o: b.c $(CC) -c -o
3. //Change
4. $(OUT)/b.o: b.c a.c $(CC) -c -o
5. //Inferring Actual build dependencies for Example-master
6. If b.o is not in the actual build dependency graph of incremental builds
7. Tracing make b.o
8. Creating actual build dependency[b.o]
    
```

Figure 6: Inferring Actual Build Dependencies for Changes to Makefile

changed. The project “Example-master” removed a file and added two files (lines 2–4). ECHECKER infers the “src/options.o” and “b.o” in the actual build dependency (lines 6–11). The file “src/Readme” is not a source code file, ECHECKER will ignore the addition of the file “src/Readme” (line 4).

Build command change analysis. ECHECKER instructs GNU MAKE to output the build commands for all build targets (using command “make -n -B -debug=basic”) after applying the commit and to use all build commands as input for the next detection. ECHECKER uses a lightweight comparison to analyze whether the build commands of targets have changed. ECHECKER identifies whether the target whose build command was changed was rebuilt during the previous incremental build. If not, the build target is rebuilt separately to obtain its actual build dependencies. Figure 6 illustrates how ECHECKER infers the actual build dependencies when the Makefile is changed. The prerequisite “a.c” (lines 1–5) is added to the target “\$(OUT)/b.o”. ECHECKER checks if “b.o” is in the actual build dependency graph of incremental builds. If not, ECHECKER rebuilds “b.o” and traces the build process to obtain its actual build dependencies (lines 6–8).

3.2.3 Inferring Actual Build Dependency Graph. ECHECKER infers a new actual build dependency graph by leveraging the results of the monitoring incremental builds execution stage and the static code analysis stage, and the historical actual build dependency graph of clean builds (step ⑤).

ECHECKER traverses all the targets from the previous two stages. For all targets in both graphs, ECHECKER checks in the historical actual build dependency graph of clean builds for targets, and updates them if they exist. If targets do not exist in the history actual build dependency graph of clean builds, ECHECKER adds them to the history actual build dependency graph. After a traversal, ECHECKER obtains the new actual build dependency graph.

3.3 Detecting Build Dependency Errors

ECHECKER detects dependency errors by leveraging the actual build dependency graph and the declared build dependency graph. In this subsection, we illustrate how ECHECKER obtains a declared build dependency graph and detects dependency errors (step ⑥).

Obtaining declared build dependency graph. ECHECKER instructs GNU MAKE to output its internal database (command “make -p”) [16] to obtain the declared build dependencies. GNU MAKE parses the Makefile during the build and stores the resulting declared build dependencies in its internal database. ECHECKER obtains the declared build dependency graph by parsing the internal database of GNU MAKE (line 1 in Figure 7). GNU MAKE will output *Phony Targets*, which are targets that do not represent actual targets (e.g., phony target “all”), but are simply a mechanism for organizing and managing target dependencies [16]. Typically, the name of a phony target is user-defined and declared in the “.PHONY” variable in Makefile. Phony targets in the declared build dependency graph will cause false positives (reporting a dependency error as a dependency error when it is not), hence ECHECKER identifies the phony targets and replaces them with the value of the “.PHONY” variable (the real targets) to avoid reporting false positives.

```
1. args.o: a.o b.o c.o
2. # Implicit rule search has been done.
3. ...
4. $(CXX) $(CXXFLAGS) $(OBJ) -o
```

Figure 7: Data Structure of the Internal Database

Detecting build dependency errors. ECHECKER detects build dependency errors by comparing the actual build dependency graph and the declared build dependency graph. For the same targets, MDs are in the actual build dependency graph, but not in the declared build dependency graph. RDs are the dependencies in the declared build dependency graph that do not appear in the actual build dependency graph.

4 ILLUSTRATIVE EXAMPLE

In this section, we use a concrete example to illustrate how ECHECKER infers the actual build dependencies and thus detects dependency errors in the *fzy* project, a fast, simple fuzzy text selector for the terminal. Commit 28195b3 from the *fzy* project has two MDs (“src/match.h” and “src/tty.h”) in the target (“src/fzy.o”).

Current state. Assume that ECHECKER is currently obtaining the actual build dependency graph from the parent commit (commit ID: 28195b3) since the practitioner has been using ECHECKER to check the past commits. If not, ECHECKER would obtain it through a clean build. ECHECKER first obtains the historical actual build dependencies of all targets through the historical actual build dependency graph.

Monitoring incremental builds execution. ECHECKER applies the changes from the commit (commit ID: f061893, as shown in Figure 8), executes an incremental build, and traces it (step ① in Figure 3). ECHECKER identifies each file output by GNU MAKE and each file input based on file system calls, obtaining the actual build dependency graph of incremental builds between the files (build targets). As Figure 9 shows, following the process (ID 174), the process (ID 174) reads the “src/fzy.c” and “src/fzy.h”, and generates “src/fzy.o”. Therefore, ECHECKER creates a node named “src/fzy.o”, which has two edges (“src/fzy.c” and “src/fzy.h”).

Pre-processor directives and Makefile change analysis. ECHECKER determines whether to update the actual build dependencies of the targets by analyzing the pre-processor directives in the source code and the build commands in the Makefile ((steps ②–④ in Figure 3)). The idea of ECHECKER is to build on top of the actual build dependency graph of incremental builds and then complete the actual build dependency graph with code analysis. ECHECKER first identifies the pre-processor directives changes of commits in source

Parent 28195b3 Commit f061893

```
1. - CFLAGS+=-Wall -Wextra -g -std=c99 -O3 -pedantic -ldeps -Werror=vla
2. + CFLAGS+=-MD -Wall -Wextra -g -std=c99 -O3 -pedantic -ldeps -
   Werror=vla
3. - rm -f fzy test/fzytest src/*.*.o deps/*.*.o
4. + rm -f fzy test/fzytest src/*.*.o src/*.*.d deps/*.*.o
5. +include $(OBJECTS:.o=.d)
```

Figure 8: Commit from *Fzy* Project

Process ID 174 Reads: “fzy/src/fzy.c”, “fzy/src/fzy.h”
 Process ID 174 Creates: “fzy/src/fzy.o”

Figure 9: Tracing Build from Fzy Project

Commit ID: f061893
 Must remake target “src/fzy.o”
 cc -DVERSION=“1.0” -D_GNU_SOURCE -MD -Wall -Wextra -g -std=c99 -O3 -pedantic -Ideps -Werror=vla -c -o src/fzy.o src/fzy.c

Commit ID: 28195b3
 Must remake target “src/fzy.o”
 cc -DVERSION=“1.0” -D_GNU_SOURCE -Wall -Wextra -g -std=c99 -O3 -pedantic -Ideps -Werror=vla -c -o src/fzy.o src/fzy.c

Figure 10: Changes of Build Command from Fzy Project with New Command Highlighted in Red.

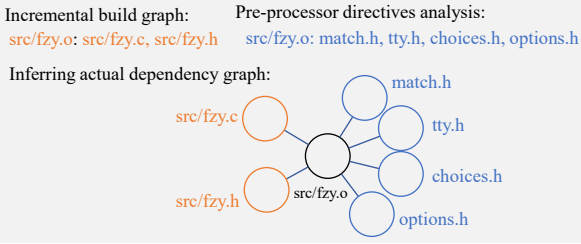


Figure 11: Inferring Actual Build Dependency Graph

files. If new pre-processor directives occur, it infers the new pre-processor directives as new actual build dependencies. Otherwise, ECHECKER considers the actual build dependencies of the target to be consistent with the historical actual build dependencies. In addition, ECHECKER analyzes the files for changes (e.g., deletions or additions). No file changes are involved in this example. More details are presented in Section 3.2.

Second, ECHECKER identifies whether the build commands in the Makefile have changed. It instructs GNU MAKE to rebuild all targets whose build commands have changed and to monitor the execution of the build for the actual build dependencies. Figure 10 shows that the “src/fzy.o” build commands have changed since it was applied to the commit. ECHECKER will re-build “src/fzy.o” and obtain the actual build dependencies for the new “src/fzy.o”.

Inferring actual build dependency graph. As shown in Figure 11, the actual build dependency of “src/fzy.o” in the actual build dependency graph of incremental builds has two edges (“src/fzy.c” and “src/fzy.h”). ECHECKER infers the actual build dependencies by analyzing the pre-processor directives of the source files affected by the current commit (step ⑤ in Figure 3). By analyzing the pre-processor directives of “src/fzy.c” and “src/fzy.h”, ECHECKER considers the actual build dependencies the same as historical actual build dependencies. The historical actual build dependencies of “src/fzy.o” also contain “match.h”, “tty.h”, “choices.h”, and “options.h”. Therefore, ECHECKER infers that “src/fzy.o” actually depends on “src/fzy.c”, “src/fzy.h”, “match.h”, “tty.h”, “choices.h”, and “options.h”.

Detecting build dependency errors. ECHECKER detects dependency errors by leveraging the actual build dependency graph and the declared build dependency graph (step ⑥ in Figure 3). It instructs GNU MAKE to output declared build dependencies and check that

the actual build dependency of each target matches the declared build dependency; any mismatch will be reported as a dependency error. For example, “src/fzy.o” has six actual build dependencies (cf. Figure 11), but the declared build dependencies of “src/fzy.o” only have “src/fzy.c”, “src/fzy.h”, “choices.h”, and “options.h”. Thus, “match.h” and “tty.h” are MDs of “src/fzy.o”.

5 EVALUATION

In our evaluation, we seek to explore the effectiveness and efficiency of ECHECKER in real-world software projects. Buildfs [44] is the state-of-the-art available method for detecting build dependency errors. Buildfs detects MDs and ordering violations (not specify ordering constraints between two dependent tasks) for C projects, but not RDs. For an apples-to-apples comparison, we configure Buildfs to detect only MDs and not execute ordering violations. We are not concerned with ordering violations as same with other methods [12, 52]. We propose the following questions (Q).

- Q.1: How effective is ECHECKER compared to Buildfs in detecting build dependency errors?
- Q.2: How efficient is ECHECKER compared to Buildfs in detecting build dependency errors?

To answer these questions, we used ECHECKER and Buildfs to detect build dependency errors in 12 projects with 240 commits. We used Buildfs’ results of detecting build dependency errors and ground truth as the baselines. For effectiveness, we evaluated whether ECHECKER can detect as many or even more build dependency errors as Buildfs (Q.1.1). We calculated precision, recall, and F-1 score for each approach based on ground truth (Q.1.2). In addition, we reported our detection results to the maintainer of projects and evaluated whether the results of our detection were accepted by the maintainers (Q.1.3). For efficiency, we compared the time consumption of ECHECKER and Buildfs for each stage of dependency error detection. Note that for a fair comparison, the time consumption of ECHECKER includes the time to obtain a historical actual build dependency graph of clean build. In addition, we established two baselines: (1) the first baseline is all 200 commits, and (2) the second baseline is 68 commits with changes to build scripts and pre-processor directives. Since a commit does not generate a new dependency error if practitioners do not modify build scripts or pre-processor directives, it is not necessary to detect build dependency errors in the commit. We manually checked each commit on the first baseline (240 commits) to obtain the second baseline (68 commits). In the second baseline, we did not include *OpenCV* and *redis* because Buildfs cannot detect build dependency errors in them.

5.1 Experimental Settings

Subjects. We selected 12 projects based on the following three characteristics: they use GNU MAKE or CMAKE as the build system, they are highly mature and popular projects, and have a significant project size (large (>1000) / medium (100–1000) / small (<100) number of files) [12, 52]. From these projects, we chose 20 consecutive commits, meaning that we overall obtained 240 commits. We randomly selected a commit as the first of 20 commits and selected the next 19 commits in chronological order. The first commit was

Table 1: Important Information about Subjects

System	Name	Stars	Files	Commits	Commit ID Range
GNU MAKE	fzy	6.1k	16	453	eb4cbd4–f061893
	chibicc	7.4k	77	316	c0f0614–90d1f7f
	ck	2.2k	501	1,654	dac27da–7eff0db
	clib	4.5k	250	434	c7c5ff6–4390598
	redis	59.5k	1,465	11,679	d2d6bc1–395d801
	python	50k	4443	115,763	3c87a66–b4612f5
CMAKE	libco	7.4k	28	103	580a446–dc6aafc
	cJSON	8.3k	228	1,073	d44b594–c680fae
	fastText	24.1k	526	381	231b871–3697152
	gravity	4.2k	1,247	764	ee4a0b0–0feea4b
	cppcheck	4.5k	1,248	26,579	e21dca2–dc0b352
	OpenCV	72k	10,644	33,466	b3d3acf–c96f48e

replaced if there were fewer than 19 commits following the first commit. Important information about subjects is shown in Table 1.

Approach application methodology. All subjects are built using their default configuration. After executing the pre-processing of the project (e.g., `./autogen`, `./configure`, and `./cmake`), Buildfs and ECHECKER check each project separately. To ensure that Buildfs and ECHECKER do not interfere with each other, we executed them in separate environments (in separate docker containers). For ECHECKER, we first performed a clean build on the first commit (e.g., b3d3acf in *OpenCV*) to obtain the actual build dependency graph and included this time in the ECHECKER’s time consumption. We then performed incremental builds for detecting dependency errors by applying changes from the new commits in ECHECKER detection. For Buildfs, we performed a clean build to detect dependency errors in each clean environment. In addition, we reported the dependency errors detected in nine projects to the maintainers, of which we obtained positive feedback from the maintainers of two projects (*cppcheck*, *cJSON*) who confirmed our reports.

Ground truth. We calculated the ground truth based on build procedures and user declarations for each subject with the default configuration. First, we parsed the build script for declared build dependencies and traced the build procedures to obtain the actual build dependencies. The declared build dependencies that do not match the actual build dependencies are identified as MDs. Moreover, for further validation of MDs, we modified the timestamps of the MDs one by one and triggered the incremental builds. There are true MDs which are expected files (the build targets of MDs) that are not rebuilt. The reason for this is that modifications to MDs do not cause a rebuild of the target file to occur [26]. Based on these, we obtain the ground truth for MDs. For RDs, we systematically and automatically removed non-actual build dependencies from the declared build dependencies for each build target and then built the target individually. The dependencies are true RDs if the target can still be successfully built after the dependency removal. Since RDs are not required for building the targets, they are just the dependencies of the practitioner’s redundant declarations in Makefile [26]. Therefore, removing them will not affect the target building process. We consider a true positive (TP) if ECHECKER agrees with the ground truth, a false positive (FP) if it does not, and a false negative (FN) if the error is not reported by ECHECKER.

Table 2: Detecting Dependency Errors in Projects. Buildfs Does not Support Detecting RDs.

Subject	Buildfs (FPs) MDs	ECHECKER		Confirmation	
		(FPs) MDs	(FPs) RDs	MDs	New
chibicc	0	0	0	0	0
cJSON	1	(16) 17	(16) 16	1	0
ck	(169) 385	216	0	216	0
clib	94	94	0	0	0
cppcheck	18	18	0	18	0
fastText	35	35	6	35	0
fzy	17	17	1	17	0
gravity	287	287	0	287	0
libco	36	36	0	36	0
redis	n/a	0	0	n/a	n/a
python	(18) 727	915	640	709	206
OpenCV	n/a	0	0	n/a	n/a
Sum	(187) 1,600	(16) 1,635	(16) 663	1,413	206

FPs: False Positives

Nevertheless, no previous study in this area ever provide a ground truth [7, 12, 26, 44, 52].

Experiment environment. Experiments were conducted on a Linux server running Ubuntu 22.04 with an Intel(R) Xeon(R) Platinum 8163 CPU@2.50GHz, 96 cores, and 376GB of physical memory. To eliminate experimental bias due to variations in device performance, we averaged the time consumed on detecting dependency errors of the two approaches by performing each detection twice.

5.2 Evaluation of Effectiveness (Q.1)

The results of detected errors are shown in Table 2. The six columns show the project name, the number of MDs and RDs detected by the two approaches (Buildfs and ECHECKER), the number of confirmed (overlapped) MDs, and the new dependency errors found by ECHECKER. In the “Confirmation” column, we listed both the same (“MDs”) and different (“New”) build dependency errors between Buildfs and ECHECKER. In the column, we excluded false positives of ECHECKER. For example, in the *cJSON* project, we found 17 MDs, 16 of which were false positives of ECHECKER. Therefore, after excluding the false positives, ECHECKER found one error, the same as Buildfs and 0 error reports were newly found build dependency errors in the *cJSON* project.

Results and true positives. ECHECKER detects 1,635 MDs and 663 RDs in 12 projects with 240 commits. Buildfs detects 1,600 MDs. The overlap in terms of actual errors between ECHECKER and Buildfs is 1,413 MDs, with 206 new MDs found. ECHECKER confirms the MDs detection results from Buildfs on 180 commits in 7 projects. ECHECKER can detect more dependency errors than Buildfs in 4 projects with two projects (*OpenCV* and *redis*) in which Buildfs cannot detect errors. Based on the ground truth of the subjects, ECHECKER has 32 false positives and 0 false negatives. On the contrary, Buildfs has 187 false positives and 206 false negatives. The precision of ECHECKER is 99.1% and that of Buildfs is 88.3%. The recall of ECHECKER is 100% and that of Buildfs is 75.7%. The F-1 score for ECHECKER is 0.995 and for Buildfs it is 0.815. ECHECKER

```

Commit ID: c69134d
1. - LIBVERSION = 1.7.9
2. + LIBVERSION = 1.7.10
#links .so -> .so.1 -> .so.1.0.0 #cJSON
3. $(CJSON_SHARED_SO): $(CJSON_SHARED_VERSION)
   ln -s $(CJSON_SHARED_VERSION)
4. $(CJSON_SHARED_SO) $(CJSON_SHARED):
   $(CJSON_SHARED_SO) ln -s $(CJSON_SHARED_SO)
   $(CJSON_SHARED)
  
```

Figure 12: False Positives of cJSON Project

improves on Buildfs by 10.8%, 24.3%, and 0.18 in terms of precision, recall, and F-1 score, respectively.

False positives. ECHECKER reports 32 false positives (16 MDs and 16 RDs) in cJSON (commit ID: c69134d, 09ebae8, 93688cb, 687b1a2). The key reason for the false positives is the improper use of the soft link command in the build command of the target (as shown in Figure 12). A deeper analysis of the false positives reveals that the issue occurs when the target “libcjson.so.1” is being built. The build command is “ln -s”, which builds “libcjson.so.1” via a soft link to “libcjson” (lines 3 and 4). The build command uses GNU MAKE to build “libcjson.so.1” by soft linking “libcjson.so.1.7.09”. However, when “libcjson.so.1” is present, the build command “ln -s” causes GNU MAKE not to retry creating the soft link. This means that when the “LIBVERSION” is changed (e.g., lines 1 and 2), the incremental builds do not complete properly. Thus, ECHECKER creates an incorrect actual build dependency graph. However, since the declared build dependencies in the Makefile have been changed in the new version, the wrong actual build dependency graph leads to false positives for MDs on the old linked source file (“libcjson.so.1.7.09”). As a result, ECHECKER produces false positives for the new linked source file (“libcjson.so.1.7.10”) and redundant dependencies on new versions of linked sources (“libcjson.so.1.7.10”). A fix is to change the link command to “ln -sf” so that the file will be regenerated even if the soft-link file already exists. In other commits in cJSON project, no further false positives are generated as the version “LIBVERSION” has not changed.

False negatives. ECHECKER does not generate false negatives in the configuration being tested. ECHECKER can only detect dependency errors in the current build configuration and will miss dependency errors in other build configurations. Practitioners usually build different software variants through software configuration. The solution is to detect software variants sequentially. Buildfs’ false negatives result from the inability to properly perform conditional compilation and correctly identify the build target (e.g., build target “Modules/grpmodule.o” in python). Buildfs obtains build targets by correctly simulating build execution, but for many projects correctly simulating build execution before a clean build is not feasible (e.g., python). In addition, Buildfs is unable to detect redis and OpenCV. It fails to obtain the build targets for these two projects since it cannot simulate the execution of these two projects due to the lack of pre-order artifacts. Thus, Buildfs is unable to detect dependency errors in the two projects.

False positives in Buildfs. We observed that for four projects, Buildfs can detect different errors from the errors reported by ECHECKER. We manually analyzed these reports. Buildfs can detect

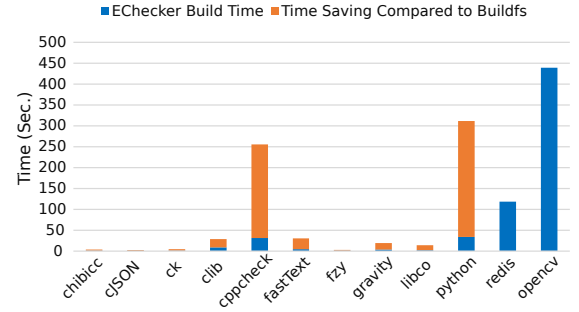


Figure 13: Average Time Saving Compared to Clean Builds

more errors in the ck project. We find that Buildfs reports dependency errors from phony targets (i.e., phony target “all”), which are false positives. Phony targets are characterized by the fact that they do not create a file, but only execute certain commands [36]. The phony target “all” specifies which targets (files) should be built.

Answer to Q.1: According to the ground truth, ECHECKER detects more dependency errors than Buildfs (Q.1.1). ECHECKER improved its F-1 score compared to Buildfs by 0.18 (Q.1.2). The maintainers of the two projects accepted our error reports (Q.1.3), which demonstrates the effectiveness of ECHECKER.

5.3 Evaluation of Efficiency (Q.2)

We have evaluated the efficiency of ECHECKER and Buildfs by measuring the time required to detect errors in the aforementioned 12 projects with 240 commits (as shown in Table 3). For an apples-to-apples comparison, we allowed Buildfs to detect only build dependency errors. The workflow of ECHECKER and Buildfs consists mainly of tracing builds and checking dependencies. Therefore, the efficiency in the evaluation consists of two components, namely build time and dependency error analysis time. Table 3 shows the build time monitored by ECHECKER and Buildfs. The table includes a comparison of ECHECKER and Buildfs in terms of tracing time, detecting build dependency errors time, and sum time. In Table 3, each value is derived from the median and sum of the 20 commits detected in each project.

Results from 240 commits. With the benefits of incremental builds and inferring actual build dependencies, ECHECKER significantly outperforms Buildfs in terms of run-time performance. For example, Buildfs for python takes about a median of 314.72s and a total of 6,268.88s, while ECHECKER takes only a median of 16.46s and a total of 716.19s. ECHECKER eliminates the need to perform multiple clean builds, thus significantly improving the efficiency of detecting dependency errors for large-scale projects. In addition, ECHECKER relies on incremental builds with the project that do not modify the program structure and are therefore more pervasive. Almost all projects benefit from the incremental build system since it is an advanced feature of a mature and stable build system.

Time savings for the build. To quantify the time saving for the build, we measure the time taken by ECHECKER and Buildfs to execute the build individually. To ensure fairness, each build is executed twice independently in a clean environment. Clean builds

Table 3: Time Consumption on Detecting Build Dependency Errors in 240 Commits (Sec.)

Projects	Buildfs						ECHECKER										Sum Difference	
	Trace		Detection		Sum		Trace		IBGraph		Inferring		Detection		Sum			
	Med	All	Med	All	Med	All	Med	All	Med	All	Med	All	Med	All	Med	All	Med	All
chibicc	3.46	68.51	0.02	0.43	3.48	68.94	0.68	25.7	0.03	0.71	0.07	2.18	0.01	0.12	0.79	28.71	2.69	40.23
cJSON	1.72	33.73	0.02	0.49	1.74	34.22	0.42	7.9	1.05	21.06	0.07	1.27	0.01	0.12	1.60	30.35	0.14	3.87
ck	4.68	94.26	0.10	1.91	4.78	96.17	0.24	10.37	0.03	0.54	0.03	2.23	0.01	0.20	0.40	13.34	4.38	82.83
clib	29.22	576.02	0.25	4.96	29.48	580.97	0.12	165.43	0.06	5.16	3.25	64.92	0.05	1.00	3.64	236.51	25.84	344.46
cppcheck	257.22	5,115.17	0.12	2.34	257.34	5,117.51	0.11	623.73	0.06	1.56	0.11	4.88	0.09	2.04	0.39	632.21	256.95	4,485.30
fastText	30.16	599.11	0.03	0.54	30.19	599.65	0.04	90.3	0.02	0.69	0.01	4.12	0.02	0.44	0.16	95.55	30.03	504.10
fzy	2.70	54.07	0.02	0.36	2.72	54.44	0.57	14.05	0.02	0.57	0.06	6.76	0.01	0.12	0.72	21.50	2.00	32.94
gravity	19.14	383.52	0.05	0.91	19.18	384.43	1.09	77.52	0.26	5.28	0.05	8.63	0.02	2.12	2.20	93.55	16.98	290.88
libco	13.51	269.49	0.03	0.51	13.53	270	1.04	32.16	0.14	2.88	0.12	2.84	0.03	0.56	1.32	38.44	12.21	231.56
python	312.94	6,233.32	1.78	35.57	314.72	6,268.88	15.20	669.16	0.53	22.54	0.01	15.07	0.44	9.42	16.46	716.19	298.26	5,552.69
redis	n/a	n/a	n/a	n/a	n/a	n/a	106.95	2,360.85	0.54	13.29	2.80	57.77	0.14	3.10	110.43	2,435.01	n/a	n/a
OpenCV	n/a	n/a	n/a	n/a	n/a	n/a	24.99	8,992.86	4.44	102.47	9.51	180.32	16.52	330.99	55.11	9,606.64	n/a	n/a

IBGraph: actual build dependency graph of incremental builds. Med: Median

and incremental builds are executed independently for each project. Figure 13 shows the time saving of the 12 projects. The results show that ECHECKER using incremental builds can significantly reduce build time, in particular for large projects. For example, in the large-scale project (*python*), incremental builds saved a median of 297.74s ($312.94 - 15.2 = 297.74$), which is 95.1% ($297.74s / 312.94s$) of the clean build time. In addition, in small-scale projects, incremental builds are still effective in saving build time. For example, in *fzy*, incremental builds save a median of 2.13s, which is 78.9% ($2.13s / 2.70s$) of the clean build time. In *cJSON*, using incremental builds reduces build time by a median of 75.6% ($1.3s / 1.72s$).

Efficiency comparison. As shown in Table 3 and Figure 14, ECHECKER improves detection time by 85.14 times on average (with the median at 16.30 times) in comparison to Buildfs. In particular, the performance advantage of ECHECKER is pronounced on large-scale projects (more than 1,000 files), such as *cppcheck* and *python*. For *cppcheck*, ECHECKER has an average improvement of 553.20 times (median is 659.07 times) over Buildfs, with a maximum

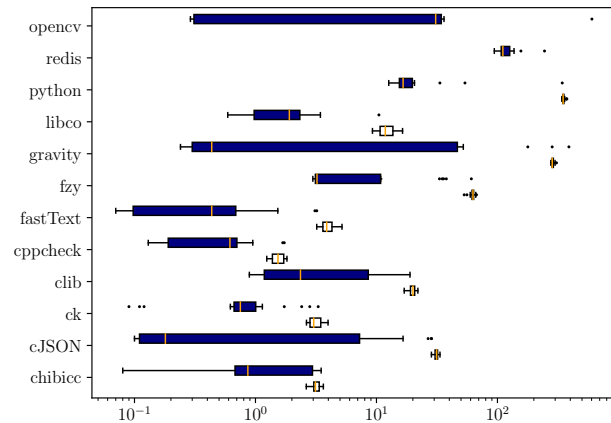


Figure 14: Sum of Time Consumption (Sec.) for Buildfs and ECHECKER to Detect in Each Commit. White Blox is Buildfs, Blue Blox is ECHECKER, and Orange Line is Median

Table 4: Time Consumption on Detecting Build Dependency Errors in 68 Commits (Sec.)

Projects	Buildfs Sum		Echecker Sum		Difference Sum		Diff*
	Med	All	Med	All	Med	All	
chibicc	3.46	37.76	0.71	17.25	2.75	20.51	0.06
cJSON	1.73	18.75	0.3	6.52	1.43	12.23	1.29
ck	4.92	38.72	3.68	29.34	1.24	9.38	-3.14
clib	29.14	197.46	2.72	75.19	26.42	122.27	0.58
cppcheck	257.61	1,280.26	0.41	269.97	257.20	1,010.29	0.25
fastText	30.11	149.42	0.54	30.97	29.57	118.45	-0.46
fzy	2.77	16.52	0.29	5.19	2.48	11.33	0.48
gravity	19.15	384.43	2.86	34.14	16.29	350.29	-0.69
libco	13.75	69.39	1.49	18.32	12.26	51.07	0.05
python	314.78	940.71	16.46	716.19	298.32	224.52	0.06

*: ECHECKER at first baseline and second baseline, the difference in median sum detection time Med: Median

improvement of 1,103.72 times (Buildfs 259.37s vs. ECHECKER 0.24s). For *redis* and *OpenCV*, Buildfs failed to perform the detection, but the clean build time for these two projects on our machine is as high as 242.32s and 8,521.85s. ECHECKER only takes 110.43s and 55.11s on the median, which demonstrates its efficiency. In addition, ECHECKER also has a smaller median than Buildfs in terms of detection time consumption. This is, because existing methods, including Buildfs, are limited by the need to use clean builds, which are time-consuming on large-scale projects. In contrast, ECHECKER supports the use of incremental builds, allowing for rapid feedback. As a result, ECHECKER completes checks in a short time. Incremental builds do not always finish quickly, in certain projects (e.g., *redis*) for instance, ECHECKER has a slightly lower efficiency. The overall execution time of incremental builds is affected by the build size required. The more artifacts an incremental build system builds, the longer it will take. In most cases, incremental builds consume less time than clean builds. Only in the worst case, does the time consumption of an incremental build equal that of a clean build. Such cases are the exceptions that do not affect the efficiency of ECHECKER.

Results from 68 commits. As shown in Table 4, ECHECKER has a higher median total detection time in seven projects than the first

baseline, ranging from 0.05s to 1.29s. The median total detection time decreased in three projects, ranging from 0.46 seconds to 3.14 seconds. Compared to Buildfs, ECHECKER increases the efficiency of error detection by 56.66 times (with a median of 5.93 times) in the second baseline.

Answer to Q.2: In the first baseline, ECHECKER detects dependency errors in 0.40s (ck)–55.11s (OpenCV), an average improvement over Buildfs of 85.14 times (median is 16.30 times). In the second baseline, ECHECKER increases the build dependency error detection efficiency by an average of 56.66 times (with a median of 5.93 times). Overall, ECHECKER’s performance decreased slightly in the second baseline compared to the first baseline, but the decrease is not significant, which demonstrates the efficiency of ECHECKER.

6 DISCUSSION

In this section, we further discuss the evaluation results, the generality, and the threats to the validity of this study.

Evaluation results. To improve the credibility of our evaluation, we went through a process to confirm the false positives and negatives of errors reported by ECHECKER. We computed the ground truth of build dependency errors based on the naive build process and user declarations. We also used the Buildfs evaluation results as the baseline. In practice, the false positive rate is an important indicator of the usefulness of the method. In total, ECHECKER detected 2,298 dependency errors, with only 32 false positives. These false positives are caused by improper build commands and can be quickly fixed by changing the build commands. There are two ways to quickly eliminate false positives when a practitioner discovers them: removing the artifact (e.g., “libcjson.so.1”) and re-detecting it or modifying the build commands (e.g., “ln -sf”). In comparison, Buildfs detected 1,600 errors, with 187 false positives (i.e., phony targets) and 206 false negatives. The lower false positive rate might make ECHECKER a more practical alternative to Buildfs.

ECHECKER has better performance in terms of the time needed to detect build dependency errors (an average improvement over Buildfs of 85.14 times). This enables ECHECKER to efficiently and consistently assist practitioners in detecting build dependency errors in the current development model of continuous integration. In contrast, the state-of-the-art methods for detecting build-dependency errors (e.g., Buildfs), which rely on clean builds, are difficult to use widely in practice due to the time consumption.

Generality. We have designed ECHECKER for C/C++ and the widely-used build tools GNU MAKE and CMAKE. However, the core methodology used by ECHECKER is not only applicable to GNU MAKE and CMAKE, it is also applicable to other build systems such as GRADLE [19], Ninja [35], and Bazel [17]. Adaptations to other build systems are required, for example, by modifying the implementation to monitor builds to adapt to GRADLE builds. As [GNU Make] builds a process for each target, while Javac does not, it builds multiple targets in a single process [25]. Other build systems provide similar functionality to simulate builds. In addition, our approach implements the use of compiler directives to read build commands and build targets in GNU MAKE. Similar functionality is also provided by other build systems. For example, Gradle provides an API [20] to achieve declared inputs/outputs of every task and

declared dependencies of every task (Gradle programmers assemble build logic in a set of tasks [18]).

Threat to validity. The main likely threat to the validity of our approach is the experimental environment of the machine, which may impact the time consumption of the experimental projects. Typically, the environment of the machine affects program execution time (e.g., build time). This affects the validity of the efficiency evaluation. To minimize the random factor, we stopped other user processes in the system, repeated the evaluation twice for individual experimental projects, and chose the average time consumption of each stage as the final result. In addition, we calculated the standard deviation of the two experiment time consumption. The results show that 2 times experiments can effectively mitigate the impact on the validity of experimental results due to fluctuations in machine performance. Since the standard deviation ranges from 0.001s (0.03% of sum detection time in *cppcheck*) to 29.771s (0.03% of sum detection time in *OpenCV*, cf. supplementary materials).

The correctness of the way we have designed to calculate ground truth might be a threat. Previous studies [7, 12, 26, 44, 52] did not consider the ground truth; their approach validated detected errors. We computed the ground truth based on the real build process and declared build dependencies. The real build process provides the ground truth for MDs. Based on this, we removed the non-missing dependencies and then built each target individually to be able to compute the ground truth of RDs. This is because true RDs do not need to build the targets.

Another potential threat is that our results might not be generalizable to other projects. Additionally, the 20 commits for each project are also not a realistic frequency of code updates for practitioners in real development. We use 12 projects with 240 commits to evaluate the effectiveness and efficiency of our approach. While this project and commit number might appear small, all of these projects are active and well-known. All of these evaluated projects are mature and widely used, demonstrating that they may be representative of significant projects with build dependency errors.

7 RELATED WORK

We subsequently discuss the most relevant research related to detecting build dependency errors.

Detecting build dependency errors. Due to the importance of build dependency errors [6, 13, 28, 38], the software engineering community has proposed many ways to detect errors. Research on detecting build dependency errors can be divided into two main categories based on whether or not it depends on the clean build.

The first category eschews clean build to detect build dependency errors. Such methods typically detect dependency errors using static analysis of source code and build scripts. Gunter et al. [21] used a Petri net to represent dependencies in build scripts to detect dependency errors in build scripts. Tamrawi et al. [47, 48] proposed SYMake to automatically detect code smells in build scripts by constructing a symbolic dependency graph. Xia et al. [53] statically analyzed build scripts and then used a link infection algorithm to infer missing dependencies. Zhou et al. [54] improved on Xia et al.’s method by using the declared build dependency graph and the source code. Both methods of Xia and Zhou were based on

MAKAO [1] and used only declared build dependencies to detect dependency errors. Their methods are efficient but ineffective because they lack actual build dependencies.

The second category relies on clean build to detect build dependency errors. Bezemer et al. [7] used MAKAO to obtain a declared build dependency graph and then analyzed the build trace log to obtain the actual build dependencies, leveraging the two dependency graphs to detect dependencies. Licker et al. [26] triggered incremental builds by modifying the timestamp of the source files and observed whether the target files had been rebuilt to detect dependency errors. However, their approach required multiple incremental builds for one detection, which was high time consumption. In addition, their approach of identifying errors by changes in the target timestamp introduced many false positives. Sotiropoulos et al. [44] treated each build target as a task, reasoning about the dependencies of each target separately, enabling it to support detecting build dependency errors in Java. Fan et al. [12] used the actual and declared build dependency graphs to propose a unified dependency graph for detecting dependency errors. Wu et al. [52] based on Fan et al.'s method proposed a virtual build per target instead of an actual build to speed up the detection of dependency errors. Their methods are efficient but lack generalizability because virtual builds require the removal of the program constructs, which can lead to build failures. Practitioners using these clean build-dependent methods have difficulty being informed in a timely manner of build-dependent errors introduced during development.

Build systems and tools. Many build systems and tools have been proposed for detecting build dependency errors [10, 11]. Rattle [45] automatically captured dependencies by monitoring the build process. Fabricate [8] and Memoize [31] automatically filled in missing dependencies for build targets by monitoring the build process at run-time. Such monitoring-based systems will inevitably slow down the build, leading to a reduction in build time. Although these build systems tried to avoid build dependency errors, they were still in their infancy and not widely used by developers due to low build efficiency, immaturity of the technology, and high migration costs. Bazel [17] creates an isolated environment for each build task and then avoids MDs by blocking the build process from accessing files that are not declared as dependencies. This avoided parallel build failures caused by missing dependencies and redundant dependencies. Other commercial tools, including IBM ClearCase [23] and VESTA [51], only checked the run-time state for dependency violations. There is a difficulty in detecting unimplemented dependencies using these methods.

In contrast to related studies, we propose a novel approach that uses code changes to infer actual build dependency graphs.

Build maintenance. There is much work involved in the maintenance of software builds [2, 3, 39–41, 50]. Gallaba et al. [14] proposed a way to decouple the acceleration of the build from the underlying build tools, speeding up the build by inferring dependencies between build steps. Build failures are common bugs. It is also urgent for developers to locate and fix the cause of the build failure. Many methods are based on historical data [22], searching [27], and deep learning frameworks [49].

8 CONCLUSION

In this paper, we have proposed a new approach, called ECHECKER, to detect build dependency errors. Unlike the existing methods [7, 12, 26, 44, 52], which rely on clean builds, ECHECKER avoids clean builds when possible to obtain the actual build dependency graph. ECHECKER uses pre-processor directives and Makefile changes to infer actual build dependencies. As a result, ECHECKER no longer requires multiple clean builds. We selected 12 representative projects with 240 commits for detecting build dependency errors to evaluate the effectiveness and efficiency of ECHECKER. The evaluation results show that ECHECKER outperforms the state-of-the-art method (Buildfs) in terms of effectiveness and efficiency, as well as demonstrate that ECHECKER has a direct impact on detecting build dependency errors for practitioners in real-world applications, as it significantly improves detection efficiency. Therefore, practitioners will be able to identify and address build dependency errors during the localization process promptly.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.62202219, No.62072227, No.62302210), the Jiangsu Provincial Key Research and Development Program (No.BE2021002-2), and the Innovation Project and Overseas Open Project of State Key Laboratory for Novel Software Technology (Nanjing University) (ZZKT2022A25, KFKT2022A09, KFKT2023A09, KFKT2023A10).

REFERENCES

- [1] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, October 2–5, 2007, Paris, France. IEEE Computer Society, 114–123. <https://doi.org/10.1109/ICSM.2007.4362624>
- [2] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault Localization for Make-Based Build Crashes. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. 526–530. <https://doi.org/10.1109/ICSM.2014.87>
- [3] Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for build code errors in makefiles. In *Proceedings of the 36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 600–601. <https://doi.org/10.1109/ICSM.2014.87>
- [4] Ant. Online. 2023. *Apache Ant Manual*. Retrieved 2023 from <https://ant.apache.org/manual/>
- [5] Autotools. 2023. *An Introduction to the Autotools*. Retrieved 2023 from <https://www.gnu.org/software/automake/manual/automake.html>
- [6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20–28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 356–367. <https://doi.org/10.1109/MSR.2017.62>
- [7] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. Germán, and Ahmed E. Hassan. 2017. An empirical study of unspecified dependencies in make-based build systems. *Empir. Softw. Eng.* 22, 6 (2017), 3117–3148. <https://doi.org/10.1007/s10664-017-9510-8>
- [8] Brush Technology. 2023. *Fabricate: The Better Build Tool. Finds Dependencies Automatically for Any Language*. Retrieved 2023 from <https://github.com/brushtechnology/fabricate>
- [9] Build Dependency. 2023. *Actual and Declared Dependencies*. Retrieved 2023 from https://docs.bazel.build/versions/main/build-ref.html#actual_and_declared_dependencies
- [10] Charlie Curtsinger and Daniel W. Barowy. 2021. LaForge: Always-Correct and Fast Incremental Builds from Simple Specifications. *CoRR* abs/2108.12469 (2021). arXiv:2108.12469 <https://arxiv.org/abs/2108.12469>
- [11] Charlie Curtsinger and Daniel W. Barowy. 2022. Riker: Always-Correct and Fast Incremental Builds from Simple Specifications. In *2022 USENIX Annual*

- Technical Conference, *USENIX ATC 2022*, Carlsbad, CA, USA, July 11–13, 2022, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 885–898. <https://www.usenix.org/conference/atc22/presentation/curtsinger>
- [12] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18–22, 2020*. ACM, 463–474. <https://doi.org/10.1145/3395363.3397388>
 - [13] Brian Fitzgerald and Klaas-Jan Stol. 2017. Continuous software engineering: A roadmap and agenda. *J. Syst. Softw.* 123 (2017), 176–189. <https://doi.org/10.1016/j.jss.2015.06.063>
 - [14] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. 2022. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Trans. Software Eng.* 48, 6 (2022), 2040–2052. <https://doi.org/10.1109/TSE.2020.3048335>
 - [15] GCC. 2023. *Using the GNU Compiler Collection (GCC)*. Retrieved 2023 from <https://gcc.gnu.org/online/docs/gcc-10.4.0/gcc/>
 - [16] GNU Make. 2023. *GNU Make Manual*. Retrieved 2023 from <https://www.gnu.org/software/make/manual/make.html>
 - [17] Google Inc. 2023. *Bazel – A Fast, Scalable, Multi-Language and Extensible Build System*. Retrieved 2023 from <https://bazel.build>
 - [18] Gradle. 2023. *Gradle User Manual*. Retrieved 2023 from <https://docs.gradle.org/current/userguide/userguide.html>
 - [19] Gradle. Online. 2023. *Gradle Build Tool*. Retrieved 2023 from <https://gradle.org/>
 - [20] Gradle Plugins. 2023. *Developing Custom Gradle Plugins*. Retrieved 2023 from https://docs.gradle.org/current/userguide/custom_plugins.html
 - [21] Carl A. Gunter. 1996. Abstracting Dependencies between Software Configuration Items. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT1996, San Francisco, California, USA, October 16–18, 1996*, David Garlan (Ed.). ACM, 167–178. <https://doi.org/10.1145/239098.239129>
 - [22] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering, 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1078–1089. <https://doi.org/10.1145/3180155.3180181>
 - [23] International Business Machines Corporation (IBM). 2023. *IBM Rational Clearcase*. Retrieved 2023 from <https://www.ibm.com/us-en/marketplace/rational-clearcase>
 - [24] Interprocess Communication Documentation. 2023. *Inter-process communication in Linux: Using pipes and message queues*. Retrieved 2023 from <https://opensource.com/article/19/4/interprocess-communication-linux-channels>
 - [25] javac. 2023. *Java Programming Language Compiler*. Retrieved 2023 from <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html>
 - [26] Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *Proceedings of the 41st International Conference on Software Engineering, 2019, Montreal, QC, Canada, May 25–31, 2019*, Joanne M. Atlee, Tefik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1234–1244. <https://doi.org/10.1109/ICSE.2019.00125>
 - [27] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we? In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, Beijing, China, July 15–19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 43–54. <https://doi.org/10.1145/3293882.3303578>
 - [28] Douglas H. Martin and James R. Cordy. 2016. On the maintenance complexity of makefiles. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics, WETSoM@ICSE 2016, Austin, TX, USA, May 14, 2016*. ACM, 50–56. <https://doi.org/10.1145/2897695.2897703>
 - [29] K. Martin. 2010. *Mastering CMake: A Cross-Platform Build System*.
 - [30] Maven. 2023. *Maven: A Software Project Management and Comprehension Tool*. Retrieved 2023 from <https://maven.apache.org/>
 - [31] Bill McCloskey. 2023. *Memoize: It is a Replacement for Make*. Retrieved 2023 from <https://github.com/kgaughan/memoize.py>
 - [32] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. 2014. Mining Co-change Information to Understand When Build Changes Are Necessary. In *Proceedings of 30th International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 241–250. <https://doi.org/10.1109/ICSME.2014.46>
 - [33] Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, May 21–28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 141–150. <https://doi.org/10.1145/1985793.1985813>
 - [34] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. 2015. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empir. Softw. Eng.* 20, 6 (2015), 1587–1633. <https://doi.org/10.1007/s10664-014-9324-x>
 - [35] Ninja. Online. 2023. *Ninja, a Small Build System with a Focus on Speed*. Retrieved 2023 from <https://ninja-build.org/>
 - [36] Phony Targets. 2023. *Phony Targets*. Retrieved 2023 from https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html
 - [37] Pre-processor Directives. 2023. *Pre-processor Directives Documentation*. Retrieved 2023 from <https://learn.microsoft.com/en-us/cpp/preprocessor/preprocessor-directives?view=msvc-170>
 - [38] Eric S Raymond. 2003. *The art of Unix programming*. Addison-Wesley Professional.
 - [39] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 71–81. <https://doi.org/10.1145/3180155.3180224>
 - [40] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. 2019. Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing. In *Proceedings of the 34th IEEE International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11–15, 2019*. IEEE, 527–538. <https://doi.org/10.1109/ASE.2019.00056>
 - [41] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. 2022. Automated Patching for Unreproducible Builds. In *Proceedings of the 44th ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25–27, 2022*. ACM, 200–211. <https://doi.org/10.1145/3510003.3510102>
 - [42] SCons. 2023. *SCons: an Open Source Software Construction Tool*. Retrieved 2023 from www.scons.org
 - [43] Hyunmin Seo, Caitlin Sadowski, Sebastian G. Elbaum, Edward Aftandilian, and Robert W. Bowdidge. 2014. Programmers’ build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 724–734. <https://doi.org/10.1145/2568225.2568255>
 - [44] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A model for detecting faults in build specifications. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 144:1–144:30. <https://doi.org/10.1145/3428212>
 - [45] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2020. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 169:1–169:28. <https://doi.org/10.1145/3428237>
 - [46] System Call. 2023. *Linux System Call in Detail*. Retrieved 2023 from <https://www.geeksforgeeks.org/linux-system-call-in-detail/>
 - [47] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. Build code analysis with symbolic evaluation. In *Proceedings of the 34th International Conference on Software Engineering, 2012, June 2–9, 2012, Zurich, Switzerland*. IEEE Computer Society, 650–660. <https://doi.org/10.1109/ICSE.2012.6227152>
 - [48] 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 International Conference on Automated Software Engineering, Essen, Germany, September 3–7, 2012*, Michael Goedicke, Tim Menzies, and Motoshi Saeki (Eds.). ACM, 366–369. <https://doi.org/10.1145/2351676.2351749>
 - [49] Daniel Tarlow, Subhdeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to Fix Build Errors with Graph2Diff Neural Networks. In *Proceedings of the 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 19–20. <https://doi.org/10.1145/3387940.3392181>
 - [50] Mohsen Vakilian, Raluca Sauciu, J. David Morgenthaler, and Vahab S. Mirrokni. 2015. Automated Decomposition of Build Targets. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 123–133. <https://doi.org/10.1109/ICSE.2015.34>
 - [51] Vesta. 2023. *Vesta Configuration Management System*. Retrieved 2023 from <http://www.vestasys.org>
 - [52] Rongxin Wu, Minglei Chen, Chengpeng Wang, Gang Fan, Jiguang Qiu, and Charles Zhang. 2022. Accelerating Build Dependency Error Detection via Virtual Build. In *Proceedings of the 37th ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10–14, 2022*. ACM, 5:1–5:12. <https://doi.org/10.1145/3551349.3556930>
 - [53] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2014. Build system analysis with link prediction. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong (Eds.). ACM, 1184–1186. <https://doi.org/10.1145/2554850.2555134>
 - [54] Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. 2014. Build Predictor: More Accurate Missed Dependency Prediction in Build Configuration Files. In *Proceedings of the IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21–25, 2014*. IEEE Computer Society, 53–58. <https://doi.org/10.1109/COMPSAC.2014.12>

Received 16-DEC-2023; accepted 2024-03-02