Detecting Build Dependency Errors by Dynamic Analysis of Build Execution Against Declaration

Jun Lyu, Shanshan Li, Bohan Liu, He Zhang, *Member, IEEE*, Guoping Rong, *Member, IEEE*, Chenxing Zhong, and Xiaodong Liu

***Abstract*—Incompletely declared build dependencies in MAKE-based build scripts can result in incorrect or inefficient incremental builds and parallel builds for C/C++ projects. In this sense, developing MAKE-based build scripts (*e.g.*, Makefile) is a nontrivial task, since practitioners need to manually enumerate the dependencies between the parts involved in one build, which may result in serious dependency errors such as missing dependencies or redundant dependencies. To tackle this challenge, the software engineering community has invested considerable effort in dependency error detection. However, due to issues such as incomplete or even missing static dependencies (*i.e.*, dependencies by users declared in Makefile), existing solutions either miss certain critical dependency errors or consume significant time when parsing build dependencies, posing a major challenge to ensure both detection effectiveness and efficiency. We propose a novel approach called BuildChecker to detect the above two critical types of dependency errors in MAKE dependencies that leverages a dynamically generated build execution-declaration model to improve error detection performance and reduce detection time. We evaluate BuildChecker with state-of-the-art tools (Mkcheck, Buildfs, VeriBuild, and VirtualBuild) on 30 projects. The experimental results show that BuildChecker is able to detect a total of 13,579 dependency errors with only 29 false positives, fewer than all the state-of-the-art tools. In terms of detection efficiency, BuildChecker outperforms Buildfs by 1.38 times and Mkcheck by 66.24 times. All dependency errors had been submitted to the practitioners and maintainers of these projects. At the time of writing this article, we received responses from the maintainers of four projects, who confirmed our error reports and fixes. BuildChecker demonstrates a great potential to support practitioners effectively detect build dependency errors.**

***Index Terms*—Build tool, build system maintenance, build dependency errors.**

Received 11 February 2024; revised 15 February 2025; accepted 16 April 2025. Date of publication 2 May 2025; date of current version 16 June 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62202219 and Grant 62302210, in part by Jiangsu Provincial Key Research and Development Program under Grant BE2021002-2, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20241195, and in part by the Innovation Project and Overseas Open Project of State Key Laboratory for Novel Software Technology (Nanjing University) under Grant ZZKT2024A18, Grant ZZKT2024B07, Grant KFKT2023A09, Grant KFKT2023A10, Grant KFKT2024A02, Grant KFKT2024A13, and Grant KFKT2024A14. Recommended for acceptance by L. Ma. *(Corresponding author: He Zhang.)*

The authors are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210093, China (e-mail: lvjun@smail.nju.edu.cn; lss@nju.edu.cn; bohanliu@nju.edu.cn; hezhang@nju.edu.cn; ronggp@nju.edu.cn; zhongcx@smail.nju.edu.cn; dz21320002@smail.nju.edu.cn).

Digital Object Identifier 10.1109/TSE.2025.3566225

I. Introduction

In the contemporary software development, automatic build systems, which can help efficiently compile and package source code into executable or deliverable artifacts, are indispensable for all projects [1], [2]. Popular build systems, such as GNU MAKE [3], NINJA [4], and ANT [5], all require build scripts to inform them the correct dependencies among build targets. Otherwise, the correctness of the build at execution time cannot be secured and may further impact the efficiency of integration [6]. Although some existing generators such as CMAKE [7] or Autotools [8] can generate build scripts automatically, for many software, practitioners have to manually enumerate all the dependencies in practice. For large projects, dependency enumeration and declaration are time-consuming, tedious, and error-prone [9].

The most common dependency-related build errors are Missing Dependencies (MDs) and Redundant Dependencies (RDs) [2], mainly due to the divergence between the static (declared) dependencies and the actual dependencies. Static dependencies are declared in build scripts by practitioners, and actual dependencies are needed during the build to produce the correct build of targets. MDs refer to the actual dependencies but not declared in the build script, while RDs refer to the declared dependencies but never used in the build.

Build dependency errors are common and easy to introduce in practice. Developers' changes to the code (addition or removal of dependencies) are not reflected in the build scripts. Seo et al. [10] examined 26.6 million builds and found that dependency errors were the most common error type in the C++ (52.68%) and Java (64.71%) projects. Kasi et al. [11] point out that changes executed in parallel by practitioners can only be reflected on build scripts after integration (merging). It is important to maintain and update the build scripts [12], [13], [14]. However, dependency management and maintenance are time-consuming and difficult [15]. It is reported that the build maintenance accounts for up to 27% of the source code development overheads and 44% of the test development overheads. Frequent manual build maintenance can affect practitioner's efficiency. The maintenance of the build is reported to significantly affect up to 79% of the source code developers and 89% of the test code developers [16].

Many studies have been conducted to help manage and maintain build dependencies by detecting dependency errors. Earlier studies used only static dependencies from the analysis of build

scripts, such as MAKAO [17], SYmake [18], or combining source code to predict MDs [19], [20]. These methods allow for efficient detection. However, due to the lack of knowing actual dependencies, the prediction results are often unsatisfactory in terms of effectiveness. Some studies try to capture actual dependencies by tracing clean builds [2], and then detect build dependency errors by updating each tested file, triggering incremental builds, and verifying that a rebuild takes place for expected files. These methods improve the effectiveness of the detection, but are less efficient. For large projects, the time taken to detect build dependency errors can be hours or even days (as shown in Table IV). Furthermore, leveraging actual dependencies and static dependencies together is another means to detect dependency errors [1], [21], [22], [23], [24]. However, existing methods do not take into account the completeness of the actual and the static dependencies obtained and therefore may generate several false positives (*e.g.*, 95 false positives within 1280 dependency errors detected [1], [23]). An existing method uses incremental builds to detect build dependency errors, which can accelerate the detection of dependency errors [24]. However, the method produces false positives due to some build commands (*e.g.*, 'ln -s'). In addition, the method requires clean-build-based detection methods (*e.g.*, the one proposed in this article) to provide a historical actual dependency graph of clean builds. Hence, the limitation of the method is whether the historical actual dependency graphs are accurate enough. To this end, obtaining a holistic view of the build process as completely as possible (*e.g.*, actual and static dependencies) is the critical issue to be addressed in dependency error detection.

In this article, we propose a new approach called BuildChecker to tackle the challenges of incomplete and inefficient build dependency error detection. The core idea is to dynamically obtain a *build execution-declaration* model that describes the specific file execution objects for each build target during the actual build process and the user-declared objects in the Makefile to detect build dependency errors. Specifically, BuildChecker obtains the build execution by monitoring any new files generated by GNU MAKE in the build and by analyzing the relationships between the process calls and the files. BuildChecker then instructs GNU MAKE to output the build declarations that are parsed by the build system. BuildChecker uses the dynamic *build execution-declaration* model that can describe the build as completely as possible to check and detect build dependency errors.

BuildChecker is evaluated using the popular and widely-used 30 open-source projects with reference to the previous studies [1], [23]. The selected projects should match the following three characteristics: their build system (GNU MAKE or CMAKE); their popularity (well-known or personal projects), and their project size (large/medium/small number of files). As the evaluation result, BuildChecker reports a total of 13,579 dependency errors with only 29 false positives, which is the most effective among all the baseline methods [2], [22] compared. Moreover, the detection efficiency of our approach outperforms Buildfs by 1.38 times and Mkcheck by 66.24 times (with reference to VeriBuild that outperforms Mkcheck by 27.80 times, VirtualBuild outperforms VeriBuild by 7.33 times).

The main contributions of this article are as follows.

- We propose BuildChecker, a more effective and efficient approach than the state-of-the-art methods to detecting build dependency errors.
- We define a *build execution-declaration* model dynamically generated to improve the effectiveness and efficiency of build dependency error detection, which enables BuildChecker to outperform the state-of-the-art methods by detecting 13,579 dependency errors with only 29 false positives within much less time.
- We discuss the main reasons for the common build dependency errors, and submit the detected errors to their practitioners with positive confirmations returned.

The outline of this article is as follows. We briefly introduce the background in Section II. Our motivations are illustrated in Section III. Section IV describes our approach to obtaining dependencies and detecting dependency errors. An evaluation of our approach is presented in Section V, followed by the discussion in Section VI. The related work is reviewed in Section VII. Finally, we conclude our work in Section VIII.

II. BACKGROUND

In this section, we introduce the background on build systems, build scripts, and incremental and parallel builds.

A. Build Systems and Build Scripts

Build systems are designed to automate the integration and compilation processes. At the heart of the automatic build is the parsing of the build scripts that declare the targets either to be built or not. MAKE is one of the most popular build systems and has been around for more than half a century [3]. After years of development, MAKE has several branches. MAKE is still by far the most widely used build system [25].

Our work focuses on GNU MAKE, which is the most widely used MAKE-based build system [26] compatible with stable Linux distributions. It obtains the dependency graph according to the build script (Makefile). When GNU MAKE build, it uses a dependency graph to determine the minimum working set. Specifically, GNU MAKE uses the dependency graph to decide whether to build from scratch or to rebuild only the modified parts.

CMAKE [7] is a widely used build system generator. One of the advantages of CMAKE is that, instead of manually writing build rules, it can automatically generate build scripts (for example, Makefile, NINJA build files), but still requires scripts (CMakelists) to work. In addition, practitioners still need to manually enumerate dependencies for custom software.

A Makefile defines many variables, macro definitions, and implicit dependencies (dependencies not declared in the Makefile, but used in build [27]). In addition, there are intermediate variables, temporary files, etc. during the build process. It is difficult to get all this information by static parsing of the Makefile. Some work attempts to parse the Makefile. MAKAO is a dependency graph inference tool that converts Makefile into a dependency graph by parsing Makefile and the command line string used to invoke the GCC compiler [17]. MAKAO

chooses to expand files with certain extensions (*e.g.*, '.c') to get implicit dependencies. The dependencies of many files without extensions are ignored in Linux. Therefore, the dependency parsing by MAKAO is incomplete. Tamrawi et al. [18] proposed a symbolic evaluation algorithm that processes Makefile and generates a Symbolic Dependency Graph (SDG) that expresses the dependencies between the build rules and the files through the build commands. It can detect code smells in the build scripts and help with renaming. Also, errors in build scripts can cause the build to fail, and some work has attempted to locate errors in build scripts. Al-Kofahi et al. [28], [29] proposed a method to locate bugs in the building code that cause failures in runtime build to help practitioners parse build scripts.

B. Incremental and Parallel Builds

Makefiles consist of a series of rules. Its typical structure is $targets : prerequisites$ [27]. Prerequisites are files that must exist before the target can be successfully built, which we refer to as dependencies[1]. Incremental builds mean that GNU MAKE will determine if the 'prerequisites' of the 'target' declared in Makefile have changed and only the changed 'target' will be rebuilt. Parallel builds determine which targets can be built simultaneously in multiple threads based on 'targets' and 'prerequisites'.

MDs and RDs may negatively affect incremental builds and parallel builds. The correct and efficient execution of incremental builds and parallel builds in GNU MAKE relies on the build script declaring the build dependencies correctly. However, MDs may lead to incorrect and inefficient builds, or '*passed*' builds but integrate incorrect parts due to the missing of correct dependencies. With MDs, changes to some files, which should have been rebuilt and relinked in incremental builds, may not trigger the correct incremental builds as expected. Thereby, the build system may link the wrong files (*e.g.*, obsolete files). Although the build system reports a '*passed*' incremental build, the software produced does not reflect the recent changes to the files overlooked in the build. Accordingly, the test cases updated for these files are very likely to fail. It is often difficult for practitioners to realize that the reason for such failed test cases is the MDs in the build script given a '*passed*' build. Hence, it may take a tremendous amount of time and effort to locate and fix such problems caused by dependency errors. Whereas RDs may introduce extra void constraints that force the sequential executions of some tasks in the build that could have been executed in parallel, which prolongs the build time. Furthermore, unnecessary incremental builds could be triggered when RDs change. In addition, all the MDs and RDs detected by BuildChecker in our experiments are from the selected projects with no build failures. The MDs and RDs detected by BuildChecker are challenging to detect and fix as they would not be indicated in the building.

As shown in Fig. 1, the user defines 5 build targets in the macro 'COLIB_OBJS' including 'co_routine.o' (line 1), and declares C source files of the format '.cpp' as dependencies

[1]If target A requires target B at build time, then A is said to depend on B, B is a dependency of A.

```
1. COLIB_OBJS=co_epoll.o co_routine.o co_hook_sys_call.o coctx_swap.o coctx.o
2. # create a list of auto dependencies
3. AUTODEPS:= $(patsubst %.o,%.d, $(COLIB_OBJS))
4. # include by auto dependencies
5. -include $(AUTODEPS)
6. %.o: %.cpp %.d $(CC) $(CFLAGS) -c -o $@ $<
7. # following were added by the authors to illustrate the RDs, not in the source code
8. co_epoll.o: lib.a …
9. co_routine.o: lib.a …
```

Fig. 1. Incremental and parallel builds errors example from *Libco* project.

(lines 2–6), but no other files (*e.g.*, header files) are included. These header files are MDs. Assuming that the practitioner modifies one of the header files, GNU MAKE does not regenerate the targets containing these header files in the incremental builds. MDs caused incorrect incremental builds. With inefficient dependency management in the Makefile (*e.g.*, RDs), only single-threaded compilation or incorrect incremental builds may be performed. It is assumed that the practitioner has declared 'lib.a' (lines 8 and 9) between 'co_routine.o' and 'co_epoll.o', but 'lib.a' is RDs. The 'co_routine.o' and 'co_epoll.o' can only be executed sequentially due to the RD (the builds of both need to wait for 'lib.a' to be generated).

The MDs studied in the field of detecting build dependency errors do not cause clean builds (full builds of a subset of software configurations in a clean environment [30]) to fail, making them more difficult for developers to detect [1], [2], [22], [23], [24]. MDs only affect incremental builds from executing correctly, *e.g.*, in Fig. 1, the absence of C source files can lead to build failures. In addition, all the MDs and RDs detected by BuildChecker in the experiments are from the included projects with no build failures. This is because, in a clean build, the compiler (*e.g.*, GCC) packages all the other files (dependencies) needed to compile the source file according to pre-processing directives (*e.g.*, #include) and adds them to the source file before executing the compilation build [31], [32], [33]. Thus, in a clean build, the build system retrieves and uses correct dependencies by itself. However, the compiler does not check pre-processing directives in incremental builds, incremental build systems only check the build script and files with new timestamps, which leads to the risk that incremental build system uses incorrect dependencies.

III. MOTIVATIONS

Previous efforts on detecting build dependency errors, such as VeriBuild [1], VirtualBuild [23], Buildfs [22], and Mkcheck [2], are reported as the state-of-the-art methods in the community.

Mkcheck shows higher effectiveness than previous studies [19], [20], [21]. Mkcheck detects build dependency errors by frequently updating the input file, triggering incremental builds, and then observing whether rebuilds occur in the expected file. While Mkcheck has been shown to improve effectiveness, it suffers the issue with low detecting efficiency. In addition, this approach results in the generation of a large number of redundant reports. The time consumption of Mkcheck in large

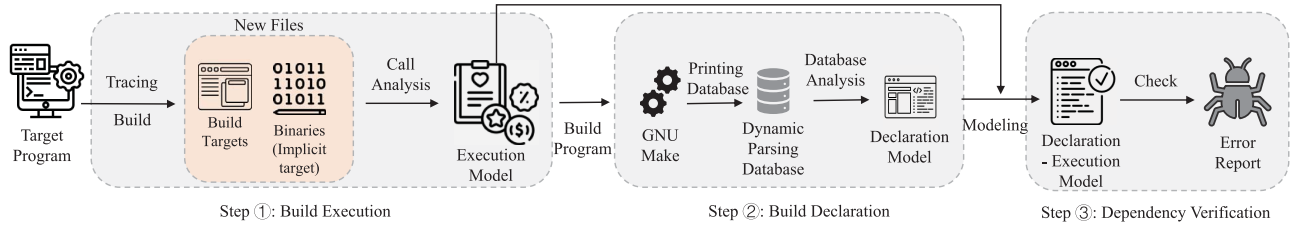

Fig. 2. Overview of BuildChecker.

projects can be as long as hours or even days (as shown in Table IV).

VeriBuild [1] and Buildfs [22] contribute to improving Mkcheck in terms of effectiveness and efficiency. VeriBuild and Buildfs solve the problem of redundant error reporting (Mkcheck issue) and further improve detection efficiency (*e.g.*, only takes about an hour for OpenCV detection). Wu et al. [23] proposed a virtual build to accelerate the detection of dependency errors in VeriBuild, which has been shown highly efficient. Their virtual build executes each target separately with modifications to the subject program structure, which still incurs effectiveness issues (*i.e.*, false positives in conditional compilation).

Buildfs relies on simulated builds to obtain build targets and dependencies. They simulate builds by instructing GNU MAKE to display the build commands that will be executed when the build task is performed, but not execute those build commands. Simulated builds cannot be applied to all projects because they cannot be executed properly without pre-order artifacts. There is an order of building between each artifact in a software build, and some of the later artifacts may check whether the earlier artifacts have been built when they are built (*e.g.*, OpenCV) [27]. VeriBuild and VirtualBuild depend on static parsing scripts. Unfortunately, static parsing scripts cannot be considered credible for obtaining static dependencies [2], because the build scripts used are not able to access all the dependencies, in which some dependencies are generated dynamically during the building process but not detectable by static parsing. These methods [1], [22], [23] violate the completeness of a software build process (the false positives on conditional compilation) and the traced targets are limited to merely the build targets. It does not trace all newly generated files that contain generated build targets as well as dynamically generated files (*e.g.*, binary files). Therefore, the '*actual dependencies*' obtained are neither the complete build executions.

Unlike existing work, BuildChecker overcomes the limitations due to static analysis scripts and simulated builds. It builds a *build execution-declaration model* for the entire software build process, rather than limiting itself to each build target, breaking through the prequel work's limitation in detecting conditional compilation. It dynamically obtains build declarations for each project instead of analyzing them statically, overcoming the inaccuracies of static analysis.

The software lifecycle can be negatively impacted by build dependency errors. Build dependency errors are difficult to resolve once and for all. It is highly probable that practitioners will continually introduce new dependency errors during their development processes. A continuous accumulation of dependency errors will eventually result in unaffordable technical debts, reducing the maintainability of software. The overall objective of our research is to devise an advanced approach for detecting build dependency errors that is capable of collecting and parsing as many complete dependencies (actual and static) as possible.

IV. Approach

In this article, we propose a novel approach, called BuildChecker, to detect build dependency errors. The core idea of BuildChecker is to dynamically generate a *build execution-declaration* model from a clean build. The model describes the specific file execution targets (files) for each build target during the actual build process and the user-declared targets (targets declared by users in Makefile [27]) in the Makefile. Specifically, instead of executing each build target individually [1], [22], [23], BuildChecker obtains the actual dependencies by tracing the complete build execution. By instructing GNU MAKE to output the user declaration, rather than statically parsing the Makefile, BuildChecker constructs the *build execution-declaration* model for detecting build dependency errors.

A. Overview

As shown in Fig. 2, to construct the actual dependencies with efficiency and correctness, BuildChecker uses the *build execution* model to describe the system calls to the underlying file system by the GNU MAKE when each new file (*e.g.*, binaries, the orange area in the figure) is created (step ①). Note that BuildChecker traces not only the build targets, but all the new files (*e.g.*, binaries) to obtain complete access to the execution. In the build declaration phase, BuildChecker obtains the declaration, which can be dynamically derived from the internal database (MAKE's internal database [27]) of GNU MAKE after the building (step ②). This way is more reliable than static parsing build scripts [1], [23] that cannot obtain the indirect rules (predefined rules in MAKE [27]), implicit dependencies, and macros expanded during the build execution [2]. In the dependency verification phase (step ③), BuildChecker builds a *build execution-declaration* model. Based on the complete and correct dependencies (actual and declared) obtained in the previous two phases, BuildChecker is able to detect build dependency errors effectively and efficiently.

```
Tracing Building
GNU Make starting a Process ID 1052
Process ID 1052 reads:
1. "/libco-master/co_routine.cpp"
2. "/usr/include/x86_64-linux-gnu/sys/select.h"
3. "/usr/include/x86_64-linux-gnu/bits/sched.h"

Process ID 1052 creats:
1. "/libco-master/co_routine.o"
```

Fig. 3. Tracing example from *Libco* project.

B. Build Execution-Declaration Model

1) Build Execution: This subsection describes in detail how BuildChecker obtains the build execution by tracing the operations of new files that are generated during the build process (step ①).

Automatic build systems (*i.e.*, GNU MAKE) generate a number of files when building software. These files contain not only build targets but also implicit targets or intermediate files (files needed somewhere on the way from source files to target files [27]), which are difficult to obtain through static parsing of the Makefile. Target-based tracing may miss some dependencies [1], [22], [23]. BuildChecker focuses on all the new files generated by GNU MAKE. Actual build dependencies are retrieved by tracing the build process executed by the build system. In a file-based build, the build is composed of a series of operations on files, *e.g.*, *read*, *delete*, and *create*. The build execution exists in the system calls [34], [35]. BuildChecker uses *ptrace* [36] to trace the actions executed by GNU MAKE on the files. In Linux, *ptrace* provides a way for the parent process to monitor and control sub-processes. *ptrace* can also modify the registers in sub-processes to realize break-point debugging and system call tracing. *ptrace* allows BuildChecker to monitor the underlying hardware and services (such as the file system). BuildChecker records all of the operations that GNU MAKE performs on files, such as *read*, *created*, *written*, *deleted*, and *renamed*. When a new file is created, BuildChecker obtains the input files needed to create the new file by analyzing the *ptrace* log. It should be noted that BuildChecker treats GNU MAKE as a black box and focuses on all the files that GNU MAKE calls when creating a new file.

Fig. 3 shows an example of tracing from the *libco* project. BuildChecker records which processes are started by GNU MAKE at build time (*e.g.*, process ID 1052 in the figure). BuildChecker traces the necessary files called by these processes to obtain the build execution. The system calls that are identified by BuildChecker include directives for file operations and directives for process communication. When the process (ID 1052) creates a new file (*e.g.*, '*/libco-master/co_routine.o*'), BuildChecker identifies which files the process called (*e.g.*, the process reads three files, including '*/libco-master/co_routine.cpp*').

2) Build Declaration: This subsection describes in detail how BuildChecker obtains the correct and complete build declarations. Typical software builds usually read build scripts (such as Makefile) by the build system (such as GNU MAKE), analyze build commands, and then execute the build.

TABLE I
DATA STRUCTURE OF GNU MAKE DATABASE

Sections	Description
Variables	Listing variables with a descriptive comment
Directories	Listing directories checked by GNU MAKE
Implicit Rules	Listing all the built-in and user-defined pattern rules in the GNU MAKE database
Pattern-specific variable values	Listing pattern-specific variables defined in the Makefile
Files	Listing custom and suffix rules related to specific files
VPATH Search Path	Listing the value of VPATH and all VPATH modes

When GNU MAKE executes the build, the build scripts are analyzed and stored in an internal database. Instead of static parsing of build scripts [1], [21], [23] or simulating build [22], BuildChecker dynamically obtains user-defined build declarations through dynamic parsing of the GNU MAKE database (*print-data-base* [27]). GNU MAKE stores the Makefiles it analyzes when executing building in its internal database, which provides greater reliability than static parsing of Makefiles. In addition, this approach overcomes the problem of non-generalizability of simulated builds, since BuildChecker reads the database after a clean build has been completed.

Obtaining build declarations by reading the internal database of GNU MAKE is more accurate than simulated builds and static analysis. Because simulated builds and static analyses do not execute the build, they cannot accurately analyze the actual execution of the build rules (*e.g.*, conditional compilation), and the effect of environment variables on the build commands (*e.g.*, building extra files). The above challenges can be solved by reading GNU Make's database during build execution.

As enumerated in Table I, the data structure of the above database can be separated into six sections, including *Variables*, *Directories*, *Implicit Rules*, *Pattern-Specific Variable Values*, *Files*, and *VPATH Search Path*. *Variables* lists variables with a descriptive comment. The directories are to be checked by GNU MAKE in *Directories*. For each directory, GNU MAKE displays the implementation details, such as device numbers, inodes, and statistics for the matching of filename patterns. *Implicit Rules* is the third section that contains all the built-in and user-defined pattern rules in the MAKE database. In addition, for the rules defined in files, their comments include the filename and line number. The *Pattern-Specific Variable Values* section lists the pattern-specific variables defined in the Makefile. A pattern-specific variable means that the effective scope of the variable definition is limited to the execution of the corresponding pattern rules.

In the case of the database, BuildChecker obtains the build declarations, often in the pattern of *target-prerequisite*. The macro definitions and variables have been fully extended. As shown in Fig. 4, the 'target' and 'not a target' are indicated, and Line 1 indicates *.l.r*, not the target. Lines 3–7 are comments, and Line 8 indicates the working target 'co_routine.o'. As shown in Algorithm 1, BuildChecker obtains all build declarations

```
1. # Not a target:
2. .l.r:
3. # Builtin rule
4. # Implicit rule search has not been done.
5. # Modification time never checked.
6. # File has not been updated.
7. # recipe to execute (built-in):
        $(LEX.l) $< > $@
        mv -f lex.yy.r $@
8. co_routine.o:co_routine.cpp, co_routine.d, co_routine.h, co_routine_inner.h,
   coctx.h
```

Fig. 4. GNU MAKE database example from *Libco* makefile.

Algorithm 1 Build Declaration

```
1: INPUT: Make database
2: OUTPUT: Declaration[]
3: Declaration = []
4: for fp in system.input do
5:     if fp start with("# files hash-table stats") then
6:         return Declaration[]
7:     else if target line pattern regular matching then
8:         target, deps = split(target line)
9:         Declaration[target] ← deps
10:    else if fp start with("# Not a target:") then
11:        Find next entry
12:    end if
13: end for
```

Execution	Target	Declaration
1. "/libco-master/co_routine.cpp"	co_routine.o	1. "co_routine.cpp"
2. "/usr/include/x86_64-linux-gnu/sys/select.h"		2. "co_routine.d"
3. "/usr/include/x86_64-linux-gnu/bits/sched.h"		3. "co_routine.h"
		4. "co_routine_inner.h"
		5. "coctx.h"

Fig. 5. Build execution-declaration model example from *Libco* project.

by traversing the output of GNU MAKE database (lines 4–6), and identifying pattern *target-prerequisite* (lines 7–11). Using the algorithm, BuildChecker obtains five prerequisites (declarations) of 'co_routine.o'.

3) Modeling Build

This subsection elaborates how BuildChecker models the build process as a *build execution-declaration* model after a clean build. By using the *build execution-declaration* model, BuildChecker is able to identify build dependency errors, *i.e.*, the differences between build execution and build declaration (such as MDs and RDs), to enable the effective detection of build dependency errors.

BuildChecker models the building process as Fig. 6. After a clean build, all the build targets (files) are described as follows:

$$Execution : Target(File) : Declaration$$

All targets (files) are modeled, even if the description or execution is empty. Fig. 5 shows an example of a concrete execution-declaration from *libco*. The build target is 'co_routine.o', the files called by 'co_routine.o' (in the orange on the left), and the build dependencies declared by 'co_routine.o' (in the blue on the right).

Algorithm 2 Build Verification

```
1: INPUT: Execution-Declaration[]
2: OUTPUT: MD_list, RD_List
3: MD_list[], RD_List[]
4: for target, deps in Execution do
5:     if deps not in Declaration[target] && deps from project DESTDIR
       then
6:         MD_list[targets]←deps
7:     end if
8: end for
9: for target, deps in Declaration do
10:    if deps not in Execution[target] && deps from project DESTDIR
       then
11:        RD_List[targets]←deps
12:    end if
13: end for
```

C. BUILD DEPENDENCY VERIFICATION

BuildChecker detects dependency errors by checking the differences in the *build execution-declaration* model for each target (file).

Algorithm 2 illustrates how BuildChecker verifies dependencies. BuildChecker checks whether the dependencies required by the target in the build execution are faithfully reflected in the build declaration (Lines 3–7). The dependencies in the build execution that are not in the build declaration are reported as MDs for the target. BuildChecker detects RDs by checking whether the dependencies of the build target in each build declaration are present in the build execution corresponding to the target (Lines 8–12). For example, in Fig. 5, BuildChecker checks the differences in the *build execution-declaration* model when detecting dependency error on the target '*co_routine.o*'. The file '*/libco-master/co_routine.cpp*' called by GNU MAKE during the build execution is correctly declared in the Makefile ('*co_routine.cpp*' in the declaration), BuildChecker recognizes '*co_routine.cpp*' as a correct dependency. Conversely, if '*co_routine.cpp*' is not declared in the Makefile, BuildChecker would identify '*/libco-master/co_routine.cpp*' as an MD.

As introduced in Section IV-B1, BuildChecker includes all the files for the build system calls in the build execution of targets. Like some basic external libraries (not part of the project itself, but a third-party library, *e.g.*, '*/usr/include/x86_64-linux-gnu/sys/select.h*' and '*/usr/include/x86_64-linux-gnu/bi-ts/sched.h*' in Fig. 3), these external libraries are not and should not be declared in the build scripts. We outline how to filter this part in the third library (Lines 4 and 9 in Algorithm 2). The project itself should be tested for dependency errors, not the external dependencies (not in the file path of the project) the project is bundled with (*e.g.*, '*/usr/include/x86_64-linux-gnu/sys/select.h*' and '*/usr/include/x86_64-linux-gnu/bits/sched.h*' in Fig. 3). In reality, it is difficult for practitioners to include all the external libraries needed in a Makefile, and some tools (*e.g.*, Autotools [8] and CMAKE [7]) are designed to help practitioners to be able to automatically generate a Makefile without having to consider basic external libraries.

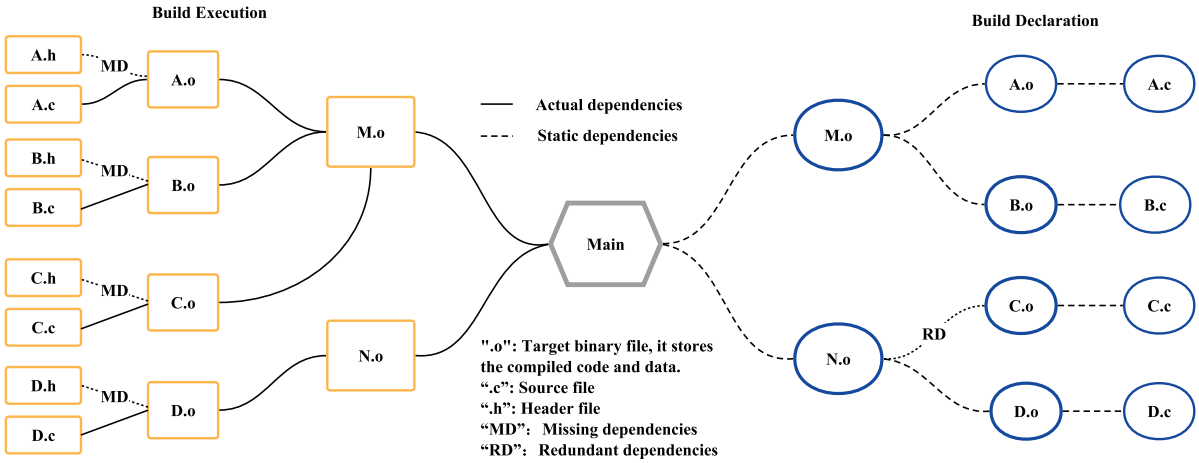

Fig. 6. Build execution & declaration.

V. Evaluation

We carried out exhaustive tests on the projects built with GNU Make and CMake to evaluate the performance of BuildChecker. An experimental evaluation is designed to answer the following research questions (RQs).

- **RQ1**: How effective is BuildChecker compared to the state-of-the-art technologies?
- **RQ2**: How efficient is BuildChecker compared to the state-of-the-art technologies?

To answer RQ1 and RQ2, we use BuildChecker and state-of-the-art methods to detect build dependency errors. To evaluate the effectiveness, with the results of the state-of-the-art methods as the benchmark, we assess whether BuildChecker could detect as many build dependency errors as the state-of-the-art tools or even more. To this end, we also design a three-stage validation method to further validate the build dependency errors reported by BuildChecker. To evaluate the efficiency, we compare the time consumed by BuildChecker against the state-of-the-art methods on detecting dependency errors.

A. Experimental Settings

Mkcheck [2], Buildfs [22], VeriBuild [1], and VirtualBuild [23] are state-of-the-art technologies for detecting build dependency errors (as shown in Table II). Compared to Mkcheck, VeriBuild improves detection efficiency and outperforms Mkcheck by 26.5 times in terms of detection time (the experiment data reported in [1], [23]). Buildfs expands the supported languages to be able to detect C-based and Java-based projects compared to Mkcheck. However, for C-based projects, Buildfs can only detect MDs. VirtualBuild [23] is a variant of VeriBuild that merely improves its efficiency. Although the direct comparison with VeriBuild and VirtualBuild would best reflect their differences in effectiveness and efficiency from BuildChecker, unfortunately, after considerable failed efforts to communicate with its development team for the intended evaluation many times, VeriBuild and VirtualBuild (developed by the identical team) are claimed to be strictly limited by some commercial policies, and declined to be accessed in public even for academic evaluation. Further for our evaluation, it becomes impossible to directly obtain the data reported in [1], [23] and also the experiment data on the same projects within the identical settings as BuildChecker for comparison. Alternatively, to the best of our effort, given Mkcheck was used as the baseline for VeriBuild and VirtualBuild and their comparison results against Mkcheck are reported in [1], [23], the evaluation had to use Mkcheck as an intermediate benchmark for the comparison between BuildChecker and the twins.

TABLE II
Build Dependency Errors Detection Methods

Methods	Support Projects	Support Build System	Errors
Mkcheck	C-based	GNU Make, CMake	MDs, RDs
Buildfs	C-based, Java-based	GNU Make, CMake, Gradle	MDs
VeriBuild	C-based	GNU Make, CMake	MDs, RDs
VirtualBuild	C-based	GNU Make, CMake	MDs, RDs
BuildChecker	C-based	GNU Make, CMake	MDs, RDs

Subjects. To answer questions, we used two subject sets. The first subject set was used to compare BuildChecker with clean build-based methods [1], [2], [22], [23]. We strove to involve the projects with reference to [1]. We use the 30 projects to evalate BuildChecker. These projects are characterized by the following features: GNU Make-based or CMake-based projects; well-known or personal projects; and different project sizes (large (>500) / medium (100–500) / small (<100) number of files). Each project is detected by all these methods with the identical version in the experiments.

Experiment equipment. All these projects are built using their default configurations. After executing Mkcheck, Buildfs, and BuildChecker separately on these projects, MDs and RDs could be detected from each project. In order to minimize fluctuations in the performance of experimental machines, each detection was executed twice [30]. The experiment was carried out on a computer equipped with an Intel(R) Core(TM) i7-12700H CPU, 2.30 GHz, 14 cores, 16GB memory, and Ubuntu 20.04.

TABLE III
COMPARISON OF DETECTED DEPENDENCY ERRORS IN PROJECTS

Name	Star	Files	BFs	VB [VrB]*		BC		MC		Confirm (BC and MC)	
			MDs	MDs	RDs	MDs	RDs	MDs	RDs	MDs\New	RDs\New
clib	4K	180	154	44	47	(8) 159	4	132	6	132\19	4\0
cJSON	7.1k	256	0	1	0	2	0	n\a	n\a	0\4	0\3
http-parser	6k	18	0	4	1	0	4	0	4	0	4\0
CacheSimulator	5	24	0	2	0	3	0	2	0	2\1	0
Stack-RNN	421	16	0	2	0	0	0	0	4	0	0
greatest	1.3k	18	0	(7) 8	0	0	0	0	5	0	0
kleaver	7	32	(1) 4	7	0	11	1	2	0	0\11	0\1
fzy	2.3k	49	0	6	0	0	0	0	14	0	0\1
libco	6.9k	30	(8) 21	20	0	13	4	n\a	n\a	0\13	0\4
cctz	475	671	0	0	0	0	0	0	0	0	0
neven	47	214	1,649	85	0	1,649	0	1,649	0	1,649\0	0
tmux	23.9k	255	1	2	0	0	0	n\a	n\a	0	0
tig	10.7k	319	0	0	0	0	2	0	4	0	2\0
ck	2k	580	395	12	1	226	0	226	(3,7518) 3,7518	226\0	0
cppcheck	4k	763	(6) 18	(39) 41	13	18	20	18	20	18\0	20\0
lec	1	323	(61) 78	2	0	2	1	2	0	2\0	0\1
redis	5.6k	805	144	(4) 11	4	8	75	144	0	8\0	0\76
8cc	5.4k	89	0	44	3	0	1	0	0	0	0\1
fastText	23.5k	524	1	11	4	(10) 11	5	1	4	1\0	4\1
gravity	4k	1,193	0	23	0	0	24	0	24	0	24\0
grbl	4.6k	70	0	16	0	0	22	0	(22) 44	0	22\0
namespaced	0	15	0	3	0	2	4	2	4	0\2	4\0
Generic-C	10	10	(2) 3	1	0	(2) 5	2	3	2	3\2	0
lighttpd1.4	1	387	0	(0) 1	0	6	3	n\a	n\a	0\6	0\3
mpc	2.3k	33	0	13 [1]	(12) 13	1	(9) 11	0	(148) 166	0\1	9\2
x86-thing	2	101	2	17	0	2	32	n\a	n\a	0\2	0\32
zlib	3.7K	293	2	(5) 6	0	1	67	1	40	1\0	0
php	34.1K	18,826	(25,386) 25,389	383	(0) 34	978	428	n\a	n\a	0\978	0\438
python	46.8k	4,545	(613) 652	(153) 158	(146) 153	748	5,689	(31,072) 86,872	(5974) 10,370	667\81	145\5,484
OpenCV	59.9k	7,269	(947) 960	(40) 68	(0) 16	282	3,053	(713) 713	(3467) 4219	0\282	0\3053
Sum	-	-	(27,114) 29,473	(37) 991	(58) 289	(20) 4,127	(9) 9,452	(31785) 89,770	(47,129) 52,448	2,708\1,399	229\9,443

BFs: Buildfs, VB: VeriBuild, VrB: VirtualBuild, BC: BuildChecker, MC: Mkcheck FP: False Positive [VrB]: False negatives of VirtualBuild
*: Data from papers [1], [23] (): False positives n\a: Unable to detect by Mkcheck

B. Evaluation of Effectiveness (RQ1)

The results of the effectiveness evaluation are shown in Table III. The columns respectively display the project name, the numbers of detected RDs and MDs by the five state-of-the-art methods—Buildfs, VeriBuild and VirtualBuild (data from [1], [23]), BuildChecker, and Mkcheck, as well as the numbers of MDs and RDs that BuildChecker confirms (overlapped) with Mkcheck and the new dependency errors discovered by BuildChecker. We confirmed that BuildChecker and MKcheck detect common errors after eliminating false positives. For example, in *clib*, BuildChecker found a total of 159 error reports, 8 of which were false positives. Therefore, after excluding these 8 error reports, there are 132 error reports by BuildChecker confirmed with the error reports detected by Mkcheck, plus 19 new error reports only discovered by BuildChecker.

Reproduction. To improve the credibility of our evaluation, we followed previous studies [1], [23] to go through a three-stage process to confirm the true positives or false positives: (1) an automatic validator was implemented to check the effectiveness of the error reported; (2) the reported errors that did not pass the automatic validation were cross-checked by three authors; (3) the checked errors were reported to the project maintainer for double confirmation. Until the time of writing this article, we received the maintainers' responses from four projects (*ck*, *cppcheck*, *clib*, and *zlib*) who accepted and fixed all of our error reports.

We start by implementing the automatic validator (1). As introduced in Section I, changes may not trigger incremental builds due to MDs, whilst RDs are unnecessarily required during target builds. To validate MDs, we modified their timestamps one by one and triggered the incremental builds. True MDs are expected files (the build targets of MDs) that are not rebuilt. For RDs, we systematically and automatically build the targets one by one and check whether the RDs are required during the build. True RDs are not required during the target builds. We also checked whether the targets with dependency errors reported by Buildfs and Mkcheck belong to the real build target declared in the Makefile. This is because phony targets (*i.e.*, phony target '*all*') can also be mistaken for real build targets.

We performed a manual test on the reports that did not pass the automatic tests (2). Due to the potential impact of subjective

TABLE IV
TIME CONSUMPTION OF DETECTION BUILD DEPENDENCY ERRORS

Project	Clean	Tracing Build Time (Sec)				Errors Detection Time (Sec)			Sum Time (Sec)				MC/VB*	VrB*/VB*
		MC	BFs	BC	Over#1	MC	BFs	BC	MC	BFs	BC	Over#2		
clib	2.89	13.45	16.34	12.29	4.65\5.64\4.25	637.54	0.04	0.19	650.99	16.37	12.48	52.15\1.31	27.90	2.52
cJSON	0.56	n\a	1.93	1.22	-\3.47\2.20	n\a	0.01	0.05	n\a	1.93	1.27	-\1.52	-	2.06
http-parser	31.21	31.83	24.28	31.99	1.02\0.78\1.02	61.61	0.01	0.07	93.44	24.29	32.06	2.91\0.76	4.91	43.41
CacheSimulator	0.36	1.56	2.13	0.88	4.36\5.94\2.44	16.80	0.01	0.05	18.36	2.14	0.92	19.89\2.31	16.65	1.77
Stack-RNN	13.58	16.49	12.53	15.69	1.21\0.92\1.16	72.76	0.01	0.08	89.24	12.54	15.77	5.66\0.79	12.27	30.15
greatest	0.73	2.79	3.60	2.98	3.85\4.96\4.11	19.27	0.01	0.08	22.06	3.60	3.06	7.22\1.18	10.78	2.16
kleaver	0.88	3.56	5.13	3.42	4.06\5.84\3.89	44.01	0.01	0.11	47.58	5.14	3.53	13.50\1.46	14.28	1.89
fzy	0.85	2.28	2.90	2.42	2.69\3.41\2.85	33.77	0.01	0.09	36.06	2.90	2.51	14.38\1.16	15.80	3.04
libco	7.25	n\a	19.30	11.08	-\2.66\1.53	57.10	0.01	0.34	n\a	19.31	11.42	-\1.69	-	4.10
cctz	15.19	25.80	37.42	24.50	1.70\2.46\1.61	125.86	0.01	0.18	151.66	37.43	24.68	6.15\1.52	8.59	8.44
neven	8.69	15.30	18.29	16.30	1.76\2.10\1.88	372.41	0.02	0.25	387.71	18.31	16.56	23.42\1.11	29.35	1.26
tmux	35.08	n\a	95.91	129.03	-\2.73\3.68	n\a	3.97	2.15	n\a	99.88	131.19	-\0.76	-	3.53
tig	10.78	32.87	52.17	26.20	3.05\4.84\2.43	644.12	0.19	0.42	676.99	52.36	26.62	25.44\1.97	35.32	3.57
ck	2.40	9.26	9.20	10.00	3.86\3.83\4.16	551.93	0.41	0.34	561.19	9.61	10.34	54.27\0.93	127.56	1.64
cppcheck	205.64	296.43	343.22	285.58	1.44\1.67\1.39	4,888.86	0.02	0.48	5,185.29	343.24	286.07	18.13\1.20	17.99	32.45
lec	20.90	55.04	89.16	50.23	2.63\4.27\2.40	244.08	0.01	0.45	299.12	89.16	50.68	5.90\1.76	7.44	-
redis	38.82	96.08	107.89	87.29	2.47\2.78\2.25	1,317.39	0.07	0.48	1,413.47	107.96	87.78	16.10\1.23	36.13	3.29
8cc	1.74	5.79	5.65	5.57	3.33\3.26\3.20	42.42	0.01	0.14	48.21	5.66	5.70	8.45\0.99	35.75	2.89
fastText	20.85	29.93	25.41	28.18	1.44\1.22\1.35	255.36	0.01	0.11	285.29	25.41	28.28	10.09\0.90	10.27	12.13
gravity	3.46	13.88	15.44	12.66	4.01\4.46\3.66	196.88	0.03	0.28	210.75	15.47	12.94	16.28\1.19	18.34	2.54
grbl	1.21	3.38	4.17	3.67	2.80\3.46\3.04	105.33	0.05	0.14	108.71	4.22	3.81	28.57\1.11	26.93	1.66
namespaced	0.48	2.41	3.24	2.16	5.05\6.82\4.54	32.74	0.01	0.11	35.15	3.26	2.27	15.48\1.43	12.56	1.39
Generic-C	0.13	0.81	0.87	0.50	6.43\6.94\3.96	10.53	0.01	0.05	11.34	0.88	0.55	20.58\1.60	11.02	1.29
lighttpd1.4	35.61	n\a	196.09	106.73	-\5.51\3.00	n\a	2.68	2.89	n\a	198.77	109.62	-\1.81	-	1.47
mpc	32.01	46.25	60.96	43.16	1.44\1.90\1.35	708.59	0.02	0.15	754.84	60.97	43.30	17.43\1.41	17.45	10.34
x86-thing	1.55	5.63	4.95	4.34	3.63\3.19\2.80	n\a	0.01	0.16	n\a	4.96	4.50	-\1.10	-	-
zlib	3.37	14.31	35.16	15.80	4.24\10.42\4.68	185.58	0.12	0.40	199.89	35.28	16.20	12.34\2.18	9.83	4.29
php	694.65	n\a	2,183.93	1,378.92	-\3.14\1.99	n\a	1.60	135.78	n\a	2,185.53	1,514.70	-\1.44	-	3.59
python	207.34	295.28	530.03	203.39	1.42\2.56\0.98	250,025.47	0.51	13.70	250,320.74	530.54	217.08	1153.11\2.44	110.21	1.45
OpenCV	4,548.08	6,773.64	7,449.65	6,645.43	1.49\1.64\1.46	278,227.71	19.17	84.14	285,001.36	7,468.81	6,729.57	42.35\1.11	49.80	16.98
AVG	198.21	311.76	378.56	305.39	2.96\3.76\2.64	21,555.12	0.97	8.13	22,775.39	379.53	313.51	66.24\1.38	27.80	7.33
Kruskal-T	-	-	-	-	-	-	-	-	-	-	-	0.0005	-	-

BC: BuildChecker, BFs: Buildfs, MC: Mkcheck, VB:VeriBuild; [VrB]:VirtualBuild false negatives *: Data from papers [1], [23] Kruskal-T: Kruskal-Wallis Test value
n\a: Unable to detect by Mkcheck; -: No data, #1: MC/Clean \ BFs/Clean \ BC/Clean #2: MC/BC \ BFs/BC

awareness on manual testing, three authors independently conducted the test, and any disagreements were resolved by a joint discussion among the authors. All uncertain dependencies were thoroughly discussed until a consensus was reached. In manual testing, we check the build declarations and build commands in the Makefile and the pre-processor directives (*i.e.*, #include [32], [33]) in the source code to determine if they are MDs or RDs. Build commands and pre-processor directives indicate which files the compiler should include in the source file for building, *i.e.*, they indicate the actual build dependencies for the build. For being identified as MDs and RDs (3), we report to the maintainer of the projects (by submitting an Issue [37]). Our error reports have been confirmed and fixed by the maintainers of four projects as of the date of writing this article.

Benchmark. We take all error reports by three tools (BuildChecker, Buildfs, and Mkcheck) after passing reproduction and de-duplicating them as the benchmark. We consider a true positive (TP) if BuildChecker agrees with the benchmark, a false positive (FP) if it does not, and a false negative (FN) if the error is not reported by BuildChecker. Our results are shown in Tables III and V. The parentheses indicate the number of false positives and brackets indicate the number of false negatives in detection.

Results and true positives. As shown in Table III, BuildChecker detects a total of 4,127 MDs (including 20 false positives) and 9,452 RDs (including 9 false positives). BuildChecker is able to detect more dependency errors than Mkcheck in 16 projects (5 of which Mkcheck cannot trace the build and detect any error). Mkcheck only triggers incremental builds by modifying some files, which makes it difficult to detect errors in other files. Both tools detected the same number of dependency errors on three projects. BuildChecker detects more dependency errors than Buildfs on 9 projects and they tie on 12 projects. Alternatively, BuildChecker can detect more dependency errors than VeriBuild on 19 projects, since VeriBuild detects dependency errors by statically parsing build scripts, which cannot locate all build targets. Moreover, VirtualBuild has a false negative based on the baseline of VeriBuild in the *mpc* project.

False positives. Based on the benchmark, BuildChecker reported 13,579 dependency errors, 29 of which were false

fastText Makefile:
fasttext: $(OBJS) src/fasttext.cc $(CXX) $(CXXFLAGS) $(OBJS) src/main.cc -o fasttext

Make data base:
fasttext:"args.o", "matrix.o", "dictionary.o", "loss.o", "productquantizer.o", "densematrix.o", "quantmatrix.o", "vector.o", "model.o", "utils.o", "meter.o", "fasttext.o", "src/fasttext.cc"

Fig. 7. False positives from *fastText*.

positives. In contrast, MKcheck and Buildfs reported 142,218 and 29,473 dependency errors but with 78,914 and 27,114 false positives respectively. As for VeriBuild and VirtualBuild, they reported 1,280 and 1,279 dependency errors respectively, and had 95 false positives each. As compared to the state-of-the-art methods, BuildChecker is capable of detecting a significant number of dependency errors with the smallest number of false positives reported. BuildChecker generates false positives only in few cases, when multiple targets are built in the same process by GNU MAKE (18 false positives in *clib* and *fastText*). It is typical for GNU MAKE to create a process for each build target. When a build system builds a target and a prerequisite (one of the precedents build targets) within the same process or its child processes of the identical parent process, BuildChecker may generate false positives. The build of prerequisites is triggered in the build of some targets in *clib* and *fastText*, then the build execution of the prerequisites is included in the build execution of the target. As shown in Fig. 7, when the target '*fasttext*' is built, it triggers the build of its prerequisites (*i.e.*, '*args.o*') in the child processes of the identical parent process. The prerequisites of '*args.o*' are included in the build execution of the target '*fasttext*', and BuildChecker reports them as MDs. However, the prerequisites of '*args.o*' should not be declared in the target '*fasttext*' and become false positives. In addition, GNU MAKE fails to output the build declarations in its entirety in a few cases (*e.g.*, 11 false positives in *Generic-C* and *mpc*), which leads to a small number of false positives in BuildChecker. Such false positives can be resolved by further parsing the build order in the build scripts. An easy-to-implement solution for false positives is to identify the actual dependencies of build targets, which are prerequisites of other targets, based on the actual build dependency graph, and to exclude those dependencies from false positives.

False negatives. With reference to the experimental results, MKcheck and Buildfs generate considerably more false negatives than BuildChecker. The results show that BuildChecker can detect 10,842 new dependency errors compared to MKcheck and 1,748 new dependency errors compared to Buildfs, which demonstrates that BuildChecker significantly reduces the number of false negatives. Note that because the original detection results of VeriBuild and VirtualBuild are unknown, we are unable to further analyze the false negatives of VeriBuild and VirtualBuild. BuildChecker cannot detect errors for build targets (false negatives) that are not executed, *i.e.*, some targets are executed in specialized software configurations. These false negatives can be resolved by executing targets that have not yet been detected.

Different dependency errors from MKcheck and Buildfs. It is noticed that in some projects the errors detected by Mkcheck and Buildfs are more than those by BuildChecker. In *ck* project alone, there are 37,518 missing dependencies errors (all of which are false positives) as well as 226 redundant dependencies reported by Mkcheck. Mkcheck produces false positives for the operations on static objects, such as modifying suffixes and copying files. Mkcheck's dependency error detection logic triggers an incremental build by modifying a file to check if a rebuild occurs for the expected file. Some build commands change the suffixes of static objects in each incremental build. Mkcheck recognizes that these files have been changed to report these files as dependency errors, which are false positives. The same reason applies to *python*. Mkcheck reports a number of errors from '_pycache_' which is automatically generated by the Python interpreter to store compiled byte code and may change during incremental builds. In addition, when Mkcheck finds dependency errors, it reports dependency errors for all the targets that contain those dependencies, even if the target does not directly depend on those dependencies[2]. Buildfs generates false positives for the files that are not monitored. These dependencies (files) can be unrecognized by Buildfs and result in false positives. In addition, Buildfs reports dependency errors for phony targets that are not represented as actual build targets (*e.g.*, phony target 'all') but are simply a mechanism for organizing and managing target dependencies [27].

> ***Answer to RQ1:*** *BuildChecker detects 4,127 MDs and 9,452 RDs in the 30 open-source projects with merely 29 false positives. Compared to state-of-the-art methods, BuildChecker is able to detect a large number of dependency errors (including a significant portion of new errors), and much fewer false positives are reported, which demonstrates its effectiveness.*

C. Evaluation of Efficiency (RQ2)

1) Comparison With Methods Based on Clean Builds: We evaluated the efficiency of BuildChecker, VeriBuild, VirtualBuild, Buildfs, and Mkcheck by calculating the time consumption of dependency error detection on the 30 projects (as shown in Table IV). Following the guidelines established by Arcuri and Briand [39], we conducted a Kruskal-Wallis Test [40] to determine the statistical significance of the differences in terms of detection time between BuildChecker, Buildfs, and Mkcheck (with $p = 0.05$). In order to minimize fluctuations in the performance of experimental machines, each detection will be executed twice. The time consumed by BuildChecker and state-of-the-art methods is primarily comprised of tracing the build and detecting dependency errors ($T_{sum} = T_{trace} + T_{detection}$). Buildfs reports only the time consumed in tracing builds and the sum of the time consumed. Table IV presents the time consumed by BuildChecker, Mkcheck, and Buildfs when tracing

[2]Direct dependencies of the target are the connected nodes with a path length of one on the actual build graph, and nodes with a length of more than one are called indirect dependencies [38].

clean builds, the extra time taken by BuildChecker, Mkcheck, and Buildfs when tracing clean builds, the time required by BuildChecker, Mkcheck, and Buildfs on detecting dependency errors, a comparison between VeriBuild and Mkcheck, as well as a comparison between VeriBuild and VirtualBuild.

Tracing build time. BuildChecker takes 0.87s (on *Generic-C*) to 6,645.43s (on *OpenCV*) on tracing build. While Mkcheck takes 0.81s (on *Generic-C*) to 6,773.64s (on *OpenCV*). While Buildfs takes 0.50s (on *Generic-C*) to 7,449.65s (on *OpenCV*). Using Mkcheck, Buildfs, and BuildChecker results in 2.96 times, 3.76 times, and 2.64 times extended build time, respectively.

Error detection time. BuildChecker takes 0.01s (on *Generic-C*) to 19.17s (on *OpenCV*) on dependency error analysis. While MKcheck takes 10.53s (on *Generic-C*) to 278,277.71s (on *OpenCV*). While Buildfs takes 0.01s (on *cJSON*) to 19.17s (on *OpenCV*).

Sum time. As shown in Table IV, detecting dependency errors using Mkcheck is a time-consuming process, taking 16.80s (on *CacheSimulator*) to 278,227.71s (on *OpenCV*). Its extreme detection time limits its efficiency and applicability in practice. BuildChecker spends 0.92s (on the *CacheSimulator*) to 6,729.57s (on *OpenCV*) on detecting dependency errors. Buildfs takes 0.88s (on *Generic-C*) to 7,468.8s (on *OpenCV*) on detecting build dependency errors.

Time saving. To quantify the time saving for the build, we measure the time taken by the three tools to execute the build individually. In terms of tracing builds, Mkcheck and Buildfs took 259.43s and 2,195.34s more time than BuildChecker, respectively (10.38s and 73.18s on average). BuildChecker improves the efficiency of tracing build by 10.8% (1 - 2.64/2.96) compared to Mkcheck and 29.8% (1 - 2.64/3.76) compared to Buildfs. In terms of detection time, Mkcheck takes 538,775.29s more time than BuildChecker, with an average of 21,551.01s. BuildChecker takes 214.86s more time than Buildfs, with an average of 7.16s. In terms of total time, Mkcheck and Buildfs take 538,976.66s and 1,980.48s longer than BuildChecker, with an average of 22,457.36s and 66.02s, respectively. The average performance of BuildChecker increases by 66.24 times compared to Mkcheck and 1.38 times compared to Buildfs.

Interpretation. BuildChecker has the advantage of tracing builds and detecting errors. Before executing a trace, BuildChecker does not require that a build be simulated (simulating a build incurs additional time). On our device, BuildChecker is 29.8% (1-2.64/3.76) faster than Buildfs in tracing builds. Furthermore, BuildChecker detects build dependency errors by comparing build execution and declaration, which is more efficient than Mkcheck's multiple incremental builds.

Answer to RQ2: *BuildChecker reduces the build time of Mkcheck by 10.8% and Buildfs by 29.8%. In terms of detecting dependency errors, BuildChecker is 66.24 times faster than Mkcheck and 1.38 times faster than Buildfs, which demonstrates its efficiency.*

VI. Discussion

In this section, we further discuss detection performance, insights into dependency error detection results, limitations, and threats to validity.

A. Detection Performance

Since VeriBuild and VirtualBuild are not accessible to the academic community, we are unable to make a valid direct comparison between the state-of-the-art methods in a unified physical environment. To the best of our effort, we have used Mkcheck as an intermediate benchmark to reflect their relative difference in performance.

Effectiveness. The difference from the previous work is that BuildChecker devises the build execution-declaration model by dynamically obtaining build declarations and build executions during the build process, and detects build dependency errors by utilizing this model. BuildChecker overcomes the challenges of analyzing conditional compilation and implicit rules that failed to be solved by the previous work, and accurately obtains build declarations and build executions by reading the GNU Make database that stores all the build declarations and build commands parsed and executed by GNU Make. In addition, BuildChecker treats software builds as black boxes, with builds monitoring the generation of all new files rather than just executing and monitoring build targets. All conditional compilations and implicit rules have been parsed and executed in a building project by GNU Make.

A wide range of dependency errors can be detected using Mkcheck, Buildfs, BuildChecker, VeriBuild, and VirtualBuild. There are different false positive rates for each tool, such as 55.5% (78,914/142,218) for Mkcheck, 91.9% (27,114/29,473) for Buildfs, 7.4% (95/1,280) for VeriBuild, 7.4% (95/1,279) for VirtualBuild, and only 0.2% (29/13,579) for BuildChecker. BuildChecker produces the lowest rate of false positives among the existing dependency error detection tools. In practice, the false positive rate is an important indicator of the practical value of a method/tool. In comparison, Mkcheck and Buildfs have a large number of redundant reports and false positives (as shown in Table III). The practical use of Mkcheck and Buildfs is limited by their large portion of false positives, requiring considerable efforts on manual double checking. Although VeriBuild and VirtualBuild reduce the number of false positives over Mkcheck and Buildfs, there are still a significant number of false positives (95 false positives). This is because the twins execute each build target individually, resulting in a large number of false positives in the case of conditional compilation. Furthermore, VeriBuild and VirtualBuild detect the fewest build dependency errors among all the state-of-the-art methods.

As a result of BuildChecker's effective build dependency detection, practitioners can quickly identify and fix errors. BuildChecker's error reports point out MDs and RDs for each build target and allow practitioners to know how to fix dependency errors (replacing MDs or removing RDs) for each build target. The error reports generated by BuildChecker are not redundant; each error report is generated as a result of a unique error. It relieves practitioners of the burden of removing redundant

TABLE V
IMPORTANT INFORMATION ABOUT PROJECTS AND DETECTING DEPENDENCY ERRORS RESULTS

Projects	Commits Range	BuildChecker		EChecker		Confirm
		MDs	RDs	(FPs) MDs	(FPs) RDs	MDs\RDs
chibicc	eb4cbd4–f061893	0	0	0	0	0\0
cJSON	d44b594–c680fae	1	0	(16)17	(16)16	1\0
ck	dac27da–7eff0db	216	0	216	0	216\0
clib	c7c5ff6–4390598	94	0	94	0	0\0
cppcheck	e21dca2–dc0b352	18	0	18	0	18\0
fastText	231b871–3697152	35	6	35	6	35\6
fzy	eb4cbd4–f061893	17	1	17	1	17\1
gravity	ee4a0b0–0feea4b	287	0	287	0	287\0
libco	580a446–dc6aafc	36	0	36	0	36\0
python	3c87a66–b4612f5	915	640	915	640	915\640
redis	d2d6bc1–395d801	0	0	0	0	0\0
OpenCV	b3d3acf–c96f48e	0	0	0	0	0\0
Sum	-	1,619	647	(16) 1,635	(16) 663	1,619\647

reports by saving their time and effort. Additionally, our future work includes automated fixing of build dependency errors based on BuildChecker's error reports.

Efficiency. According to Table IV, BuildChecker significantly improves time efficiency by 66.24 times and 1.38 times compared to Mkcheck and Buildfs. We were unable to compare the five methods in a unified environment since the maintainers of VeriBuild and VirtualBuild declined to provide licenses for such a comparison. It is claimed in [1], [23] that VeriBuild improves overall time efficiency by 27.80 times over Mkcheck, and VirtualBuild improves overall time efficiency by 7.33 times over VeriBuild. Alternatively, we had to use Mkcheck as an intermediate baseline for indirect comparison. As a result, BuildChecker is likely to improve time efficiency by 2.38 times (66.24/27.80) when compared to VeriBuild and reduce time efficiency by 3.07 times when compared to VirtualBuild ($27.80 \times 7.33/66.24$). However, BuildChecker has a 37 times lower false positive rate (0.2%) than VirtualBuild (7.4%). Considering that checking and removing false positives is a time-consuming manual task, whereas BuildChecker consumes no more than 300s in detecting dependency errors in the majority (90%) of projects (cf. Table IV), BuildChecker is able to offer comparable or higher overall detection efficiency than VirtualBuild in most cases in practice.

In summary, BuildChecker is able to detect a large number of dependency errors while generating an affordable tiny number of false positives, extremely fewer than the other state-of-the-art methods.

B. BuildChecker and EChecker

In previous work, we proposed a method called EChecker to support the detection of build dependency errors in incremental builds [24]. The core idea of EChecker is to update the actual dependency graph based on the historical actual dependency graph of a clean build by inferring actual dependency changes based on commits. Therefore, EChecker is able to detect build dependency errors in incremental builds. However, it requires clean-build-based detection methods (*e.g.*, the one proposed in this article) to provide a historical actual dependency graph of clean builds.

A comparison was made between BuildChecker and EChecker across 12 projects and 240 commits (20 commits per project) with reference to the previous study [24]. EChecker is designed to quickly detect errors in continuous code changes (multiple contiguous commits). Therefore, the subjects in Section V is not applicable as they are all single commits. Important project information and results are shown in Table V.

In the comparison, BuildChecker detects a total of 1,619 MDs and 647 RDs. EChecker detects a total of 1,635 MDs and 663 RDs, but 16 MDs and 16 RDs are false positives. EChecker cannot effectively detect the presence of special commands in a target (*e.g.*, soft links, 'ln -s'). The soft links determine if the target exists before relinking, and if it does, it is not relinked. Since EChecker uses incremental builds for detection, soft links will not relink targets. Therefore, EChecker generates false positives. In contrast, BuildChecker, because it detects build dependency errors in clean builds, is not affected by these build commands (*e.g.*, 'ln -s') and therefore does not generate false positives.

BuildChecker and EChecker are effective tools for developers to detect build dependency errors. However, they work in different application scenarios and have unique features that make them irreplaceable with each other. The focus of these two tools in addressing key issues and their respective application scenarios differ significantly. BuildChecker serves as an effective detection solution for developers, particularly in situations where code commits occur sporadically. In contrast, EChecker is designed to meet the needs of frequent code commits. It is dedicated to providing a solution that facilitates rapid error detection. By utilizing the historical actual dependency graph of clean builds, EChecker can predict changes in actual dependencies, thereby exploring a method for rapid dependency error detection. Given the distinct advantages and capabilities of BuildChecker and EChecker, developers have the flexibility to select the appropriate tool based on their specific development scenarios and diverse requirements.

ck Makefle:

```
ck_barrier_combining.o:
$(SDIR)/ck_barrier_combining.c
$(CC) $(CFLAGS) -c -o
$(TARGET_DIR)/ck_barrier_combin
ing.o
$(SDIR)/ck_barrier_combining.c
```

File ck_barrier_combining.c:

```
#include <ck_barrier.h>
#include <ck_cc.h>
#include <ck_pr.h>
#include <ck_spinlock.h>
```

Fig. 8. Missing dependencies from *Ck*.

clib Makefile

```
ifdef EXE BINS = clib.exe clib-install.exe clib-search.exe
else BINS = clib clib-install clib-search
endif
$(BINS): $(SRC) $(OBJS)
 $(CC) $(CFLAGS) -o $@ src/$(@:.exe=).c $(OBJS) $(LDFLAGS)
```

Make data base:

```
clib-install: “src/clib-install.c”, “src/clib.c”, “src/clib-search.c”,
“deps/str-replace/str-replace.o”, “deps/list/list.o”,
“deps/list/list_node.o”, “deps/list/list_iterator.o”,
“deps/parson/parson.o”, “deps/path-join/path-join.o”, …
```

Fig. 9. Redundant dependencies from *Clib*.

C. Insights Into Dependency Errors Detected

Analyzing error reports from BuildChecker and Makefiles of subjects provides some new insights. As a result of our analysis, we have found that dependency errors are often caused by a shortage of static dependency management and a violation of the build rules.

Practitioners often fail to correctly specify indirect and direct dependencies as they only specify ‘.c’ files in their build scripts because of the fact that header files are missing for the references to ‘.c’ files (resulting in an error of missing direct and indirect dependencies). In GNU MAKE, missing dependencies may not cause a build to fail. Therefore, practitioners are used to declaring only ‘.c’ files without taking into account direct and indirect dependencies. Fig. 8 shows an example of missing dependencies from *ck* project. The only dependency declared by the target *ck_barrier_combining.o* in the Makefile is *ck_barrier_combining.c*. However, BuildChecker reports many missing dependencies for this target. According to the source code of *ck_barrier_combining.c*, there are four direct references and many indirect references. These references are not declared in the Makefile, resulting in missing dependencies. It is a tedious and error-prone process to find all references. For regular dependency detection, practitioners are advised to use automated dependency retrieval tools or BuildChecker.

In addition to specifying only source files without dependencies, declaring too many dependencies can also be problematic. It is common for practitioners to define a set of targets and declare all possible dependencies for these targets. For a single target, a large number of redundant dependencies may be declared. As shown in Fig. 9, targets within each *BINS*, such as *clib-install*, are declared with a large number of dependencies, many of which are redundant. These RDs force the sequential execution of the targets that could be executed in parallel, reducing the efficiency of parallel builds. Additionally, when these targets change, unnecessary incremental builds will be triggered.

D. Limitation

Linux has hundreds of system calls, and we do not analyze every one of them. As much as possible, we profile file system-related system calls in our approach. Although some projects may use special system calls to build, all file operations in our experiment are monitored effectively. In addition, we use *ptrace* in Linux to trace the building. Similar functionality is provided in Windows (Debugging API [41]). The extension of BuildChecker to Windows requires the necessary modifications to the method of tracing the building process.

We have designed BuildChecker for C/C++ projects and the widely used build tools, GNU MAKE and CMAKE. BuildChecker’s methodology applies not only to GNU MAKE and CMAKE, but also to other Make branches and build systems such as BSD MAKE [42], GRADLE [43], Ninja [4], and Bazel [44], as well. This implementation of the build monitoring system and build declaration parsing system must be modified to accommodate different build systems. For example, for the GRADLE builds, GNU MAKE creates processes for each target, whereas Javac [45] does not. As a result, the build process builds multiple targets in the same process. Additionally, our approach incorporates the use of compiler directives to read and build targets within GNU MAKE. Other build systems provide similar functionality as well. GRADLE provides an API [46] for obtaining declared inputs/outputs of each task and declared dependencies (GRADLE programmers assemble build logic using a set of tasks [47]).

Existing tools [1], [2], [22], [23], [24] for detecting dependency errors rely on the successful execution of the build, and so does BuildChecker. It will not function if a build fails. BuildChecker is designed to identify dependency errors not detected or reported by the build system, rather than analyzing build errors. It is beyond the scope of this study to analyze build failures.

E. Threats to Validity

Internal validity. The physical experimental environment may introduce a threat to the validity of our approach since it will affect the duration of the experiment. The computing environment affects program execution time, such as build time. This impact might adversely affect the validity of the efficiency evaluation. To mitigate this threat, we halt other user processes in the system and perform detection twice for each project in order to minimize the possible impact of random factors from the environment. The experimental results were recorded by averaging the time consumed at each stage. Furthermore, we calculated the standard deviation of the time consumed by the three tools (BuildChecker, Buildfs, and Mkcheck) when performing the two detections. The results showed that the standard deviation of detection time for each tool was small. The range of standard deviations in detection time for each tool is Mkcheck 0.226s (0.5% of sum detection time)–2,160.806s (0.7% of sum detection time), Buildfs 0.001s (0.04% of sum detection time)–37.263s (0.7% of sum detection time), and BuildChecker 0.002s (0.02% of sum detection time)–41.399s (0.6% of sum detection

time). Consequently, the time consumption for each project is able to reveal BuildChecker's efficiency.

External validity. The threat is the potential generalization of the results to other projects that are not included in this work. The effectiveness and efficiency of our approach are evaluated using 30 projects. As a means of mitigating this risk and for potential comparison, the projects involved in evaluating BuildChecker are all from the projects evaluated by previous studies. Additionally, an exhaustive test of BuildChecker has been conducted on a selection of projects from multiple build systems, with a variety of prevalence and sizes. They are potentially representatives of important projects that suffer build dependency errors.

Conclusion validity. The correctness of the way we calculate the benchmark might be a threat. We calculated the benchmark based on the detection results of the three tools. It is not to be ignored that all three tools have false positives, and therefore the benchmark calculation is not completely correct. However, both Mkcheck and Buildfs are advanced tools for detecting dependency errors. In addition, we designed reproduction tools to confirm that the detected dependency errors are real dependency errors. To improve the credibility of our evaluation, we went through a three-stage process to confirm the true positives or false positives: 1) an automatic validator was implemented to check the effectiveness of the errors reported; 2) the reported errors not passing the automatic validation were then cross-checked by three authors; 3) the manually checked errors were reported to the project maintainers for double confirmation.

One of the validity threats is the correctness of our design of the automatic validator, which is based on a real build process and the definition of build dependency errors from Licker et al. [2]. In this validator, MD timestamps are modified and an incremental build is triggered. True MDs do not cause the target to be rebuilt. By building the targets individually and observing whether RDs are used, the validator accomplishes validating the RDs. Since RDs are redundant declarations in the Makefile, they are not used in the target build. Moreover, manual analysis of error reports that do not pass automated validation poses a threat to validity since the analyst's knowledge may influence the results of the manual analysis. Three authors independently conducted our manual analyses, with team discussions to resolve any differences of opinion.

Further, due to the unavailability of VeriBuild and VirtualBuild in the academic community, Mkcheck was used as an intermediate baseline to evaluate BuildChecker's performance in comparison to the twins. BuildChecker is not able to be directly compared against VeriBuild and VirtualBuild in the same environment since they declined to be accessed and evaluated in public. However, a relatively indirect comparison and evaluation of performance between BuildChecker and the twins on identical projects were conducted via Mkcheck as an intermediate benchmark.

VII. Related Work

This section briefly reviews some reported studies related to detecting software build dependency errors.

Build dependency error detection. Relevant studies can be divided into three major categories. The first category uses a static dependency graph to detect build dependencies. For example, Xia et al. [19] analyzed build profiles to predict possible missing dependencies using a link prediction algorithm. Zhou et al. [20] improved Xia's method in terms of effectiveness by leveraging the static dependency graph and source code. Both Xia's and Zhou's methods are based on MAKAO and only use static dependencies to detect dependency errors. Due to the unknown actual dependencies, the effectiveness of such methods is not ideal.

Studies in the second category, on the other side, focus on actual dependency and construct the actual dependency graph by monitoring the real software build process. Licker et al. [2] proposed a method to detect both MDs and RDs, which requires multiple incremental builds, resulting in extremely high costs on time. Furthermore, since their method only targets specific files, missing file dependencies could not be detected, resulting in limited results.

Studies in the third category leverage static dependencies and actual dependencies. In the early work, Bezemer et al. [21] used MAKAO to obtain the static dependency graph and then traced the build process through *strace* to obtain the actual dependency graph. Dependency errors can be discovered by using the two types of dependency graphs. However, MAKAO cannot find implicit dependencies for the files with a non-specific extension (*cf.* Section II-A). Hence, their method may bear the risk of missing static dependencies and producing a large number of false positives. Instead, Sotiropoulos et al. [22] used both dependency graphs to detect MDs only. Fan et al. [1] similarly used these two dependency graphs to propose a unified dependency graph that turns dependency detection into a graph traversal problem. Wu et al. [23] proposed a virtual build approach to accelerate build dependency error detection, but this approach is neither efficient enough nor applicable to all projects. Unfortunately, their methods [1], [23] rely on statically parsing the Makefile or cutting the build into per-targets, which generates many false positives. Lyu et al. proposed a method called EChecker for inferring dependencies to support the ability to quickly detect dependency errors in incremental builds [24].

Unlike these relevant studies, by tracing and parsing the build process, BuildChecker is able to obtain actual dependencies (by complete tracing) as well as static dependencies (by accurate declaration parsing), and then dynamically generate execution build model descriptions using *build execution-declaration* model introduced in this article, which results in fewer false positives in and higher effectiveness of dependency error detection as reflected in the evaluation.

Build systems on build dependency errors. The Bazel build system from Google provides functionality to detect MDs [44]. However, its user documentation emphasizes that such detection is not always possible [48]. Tup [49] and IBM ClearCase [50] can check for MDs at runtime. Some build techniques, such as Fabricate [51] and Memoize [52], avoid dependency errors because they do not require a prior declaration of build dependencies. Based on this, Spall et al. [53] implemented a novel build system, which only needs commands to build and does

not need to specify dependencies but captures dependencies by tracing execution. Their build system is inefficient for parallel builds.

Build reproducibility. Verifying build reproducibility can uncover some critical bugs. Ren et al. [54] searched for problematic files that led to an unreproducible build for the localization task. They also identified the root cause of unreproducible build by tracing the build process across different environments [55] and later searched for similar patches based on historical knowledge to fix an unreproducible build [56].

Fix build failure. It is also important for practitioners to locate and fix the cause of the build failure. Common methods are based on historical data [57], searching [58], and deep learning frameworks [59].

Build acceleration The build acceleration techniques have been widely focused on and the common techniques include, distributed builds [60], compilation caching [61], parallel builds, and incremental builds [27]. Gallaba et al. [62] 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. Randrianaina et al. [30] demonstrated that it is possible to accelerate the builds of multiple software configurations by means of multiple incremental builds. Lyu et al. [63] further proposed to develop an incremental build order for multiple software configurations in order to accelerate the builds of all the configurations.

VIII. Conclusion

Contemporary continuous software engineering [13] has to tackle the challenge of frequent changes, which requires a fast build to test the impact of code changes for instant feedback. Incremental builds and parallel builds play a vital role in supporting fast builds, for which build scripts need to declare the correct dependencies properly. In practice, build dependencies still need to be manually enumerated by practitioners in most cases, which is prone to errors.

This article proposes a novel approach, BuildChecker, to detect missing dependencies and redundant dependencies. Unlike the previous work, a dynamically generated *build execution-declaration* model is introduced in this approach to correctly describe the entire build process. BuildChecker allows for a complete and correct description of build dependencies. We evaluate BuildChecker with state-of-the-art methods (Mkcheck, Buildfs, VeriBuild, and VirtualBuild) on 30 open-source projects. The evaluation results show that BuildChecker can improve the detection efficiency by 66.24 times (Mkcheck) and 1.38 times (Buildfs), with a total of 13,579 errors detected and only 29 false positives. The dependency errors have also been reported to all of the project maintainers, and at the time of reporting, four projects have confirmed and fixed in response to our reports.

In the future, BuildChecker can be extended to other build systems. BuildChecker obtaining static dependency graphs not only applies to GNU Make and CMake at this stage. The reason for focusing on GNU Make and CMake is that early work has confirmed that GNU Make and CMake are the widely used build systems [25], [26]. The core of our static dependency extraction is data analysis. For other build systems that support build script debugging, our approach can still be valid and applicable just by updating the data structure correspondingly.

References

[1] G. Fan, C. Wang, R. Wu, X. Xiao, Q. Shi, and C. Zhang, "Escaping dependency hell: Finding build dependency errors with the unified dependency graph," in *Proc. ISSTA: 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal., Virtual Event*, USA, Jul. 18–22, 2020, pp. 463–474. [Online]. Available: https://doi.org/10.1145/3395363.3397388

[2] N. Licker and A. Rice, "Detecting incorrect build rules," in *Proc. 41st Int. Conf. Software Eng. (ICSE)*, Montreal, QC, Canada, J. M. Atlee, T. Bultan, and J. Whittle, Eds., May 25–31, 2019, pp. 1234–1244. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00125

[3] S. I. Feldman, "Make-a program for maintaining computer programs," *Softw. Pract. Exp.*, vol. 9, no. 4, pp. 255–65, 1979. [Online]. Available: https://doi.org/10.1002/spe.4380090402

[4] Ninja, "Ninja, a small build system with a focus on speed," 2012. [Online]. Available: https://ninja-build.org/

[5] Apache, "Apache ant manual," 2012. [Online]. Available: https://ant.apache.org/manual/

[6] C. Lebeuf, E. Voyloshnikova, K. Herzig, and M. D. Storey, "Understanding, debugging, and optimizing distributed software builds: A design study," in *Proc. IEEE Int. Conf. Software Main. Evol. (ICSME)*, Madrid, Spain, Sep. 23–29, 2018, pp. 496–507. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00060

[7] K. Martin and B. Hoffman, *Mastering CMake: A Cross-Platform Build System.* 2010.

[8] GNU Automake, "An Introduction to the Autotools," 2020. [Online]. Available: https://www.gnu.org/software/automake/manual/automake.html

[9] E. S. Raymond, *The Art of Unix Programming.* 2004.

[10] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: A case study (at Google)," in *Proc. 36th Int. Conf. Software Eng. (ICSE)*, Hyderabad, India, P. Jalote, L. C. Briand, and A. van der Hoek, Eds., May 31–Jun. 7, 2014, pp. 724–734. [Online]. Available: https://doi.org/10.1145/2568225.2568255

[11] B. K. Kasi and A. Sarma, "Cassandra: proactive conflict minimization through optimized task scheduling," in *Proc. 35th Int. Conf. Software Eng. (ICSE)*, San Francisco, CA, USA, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., May 18–26, 2013, pp. 732–741. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606619

[12] D. H. Martin and J. R. Cordy, "On the maintenance complexity of makefiles," in *Proc. 7th Int. Workshop Emerg. Trends Software Metrics, WETSoM@ICSE*, Austin, TX, USA, May 14, 2016, pp. 50–56. [Online]. Available: https://doi.org/10.1145/2897695.2897703

[13] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *J. Syst. Softw.*, vol. 123, pp. 176–189, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2015.06.063

[14] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis Ci with github," in *Proc. 14th Int. Conf. Mining Software Repositories (MSR)*, Buenos Aires, Argentina, J. M. González-Barahona, A. Hindle, and L. Tan, Eds., May 20–28, 2017, pp. 356–367. [Online]. Available: https://doi.org/10.1109/MSR.2017.62

[15] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *Proc. 30th IEEE Int. Conf. Software Main. Evol.*, Victoria, BC, Canada, Sep. 29–Oct. 3, 2014, pp. 401–405. [Online]. Available: http://arxiv.org/abs/1512.01862

[16] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proc. 33rd Int. Conf. Software Eng. (ICSE)*, Waikiki, Honolulu, HI, USA, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., May 21–28, 2011, pp. 141–150. [Online]. Available: https://doi.org/10.1145/1985793.1985813

[17] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter, "Design recovery and maintenance of build systems," in *Proc. 23rd IEEE Int. Conf. Software Maintenance (ICSM)*, Paris, France, Oct. 2–5, 2007, pp. 114–123. [Online]. Available: https://doi.org/10.1109/ICSM.2007

[18] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proc. 34th Int. Conf. Software Eng., (ICSE)*, Zurich, Switzerland, Jun. 2–9, 2012, pp. 650–660. [Online]. Available: https://dl.acm.org/doi/10.5555/2337223.2337300

[19] X. Xia, D. Lo, X. Wang, and B. Zhou, "Build system analysis with link prediction," in *Proc. Symp. Appl. Comput. (SAC), Gyeongju*,

Republic of Korea, Y. Cho, S. Y. Shin, S. Kim, C. Hung, and J. Hong, Eds., Mar. 24–28, 2014, pp. 1184–1186. [Online]. Available: https://doi.org/10.1145/2554850.2555134

[20] B. Zhou, X. Xia, D. Lo, and X. Wang, "Build predictor: More accurate missed dependency prediction in build configuration files," in *Proc. Comput. Softw. Appl. Conf.*, 2014. [Online]. Available: https://doi.org/10.1109/COMPSAC.2014.12

[21] C. Bezemer, S. McIntosh, B. Adams, D. M. Germán, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3117–3148, 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9510-8.

[22] T. Sotiropoulos, S. Chaliasos, D. Mitropoulos, and D. Spinellis, "A model for detecting faults in build specifications," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA2020. [Online]. Available: https://doi.org/10.1145/3428212.

[23] R. Wu, M. Chen, C. Wang, G. Fan, J. Qiu, and C. Zhang, "Accelerating build dependency error detection via virtual build," in *Proc. 37th ACM Int. Conf. Automated Software Eng. (ASE)*, Rochester, MI, USA, Oct. 10–14, 2022, pp. 1–5. [Online]. Available: https://doi.org/10.1145/3551349.3556930

[24] J. Lyu, S. Li, H. Zhang, Y. Zhang, G. Rong, and M. Rigger, "Detecting build dependency errors in incremental builds," in *Proc. 33rd ACM SIGSOFT Int. Symp. Software Test. Anal. (ISSTA)*, Vienna, Austria, M. Christakis and M. Pradel, Eds., Sep. 16–20, 2024, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3650212.3652105

[25] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Emp. Softw. Eng.*, vol. 20, no. 6, pp. 1587–1633, 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9324-x

[26] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Proc. 30th IEEE Int. Conf. Software Maintenance Evol.*, Victoria, BC, Canada, Sep. 29–Oct. 3, 2014, pp. 241–250. [Online]. Available: https://doi.org/10.1109/ICSME.2014.46

[27] GNU make, "GNU make manual," 2020. [Online]. Available: https://www.gnu.org/software/make/manual/make.html

[28] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "Fault localization for make-based build crashes," in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 526–530. [Online]. Available: https://doi.org/10.1109/ICSME.2014.87

[29] J. M. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "Fault localization for build code errors in makefiles," in *Proc. 36th Int. Conf. Software Eng. (ICSE)*, Companion Proceedings, Hyderabad, India, P. Jalote, L. C. Briand, and A. van der Hoek, Eds., May 31–Jun. 7, 2014, pp. 600–601. [Online]. Available: https://doi.org/10.1145/2591062.2591135

[30] G. A. Randrianaina, X. Tërnava, D. E. Khelladi, and M. Acher, "On the benefits and limits of incremental build of software configurations: An exploratory study," in *Proc. 44th Int. Conf. Software Eng.*, Pittsburgh, PA, USA, May 25–27, 2022, pp. 1584–1596. [Online]. Available: https://doi.org/10.1145/3510003.3510190

[31] Include Operation, "Include operation," 2024. [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/Include-Operation.html

[32] Pre-Processor Directives, "Pre-processor directives documentation," 2023. [Online]. Available: https://learn.microsoft.com/en-us/cpp/preprocessor/preprocessor-directives?view$=$msvc-170

[33] C/C++ Pre-Processor, "C/C++ Pre-Processor Reference" 2023. [Online]. Available: https://learn.microsoft.com/en-us/cpp/preprocessor/c-cpp-preprocessor-reference?view$=$msvc-170

[34] Linux, "Linux system call in detail," 2023. [Online]. Available: https://www.geeksforgeeks.org/linux-system-call-in-detail/

[35] Inter-Process Communication, "Inter-process communication in linux: Using pipes and message queues," 2023. [Online]. Available: https://opensource.com/article/19/4/interprocess-communication-linux-channels

[36] Ptrace, "Ptrace," 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/ptrace.2.html

[37] Github, "Use github issues to track ideas, feedback, tasks, or bugs for work on github," 2023. [Online]. Available: https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues

[38] Bazel Build, "Actual and declared dependencies," 2022. [Online]. Available: https://docs.bazel.build/versions/main/build-ref.html#actual_and_declared_dependencies

[39] A. Arcuri and L. C. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Software Eng. (ICSE)*, Waikiki, Honolulu, HI, USA, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., May 21–28, 2011, pp. 1–10. [Online]. Available: https://doi.org/10.1145/1985793.1985795

[40] E. Ostertagová, O. Ostertag, and J. Kováč, "Methodology and application of the Kruskal-Wallis test," *Appl. Mech. Mater.*, vol. 611, pp. 115–120, 2014.

[41] The Debugging Application, "The debugging application programming interface," 2024. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/ms809754(v$=$msdn.10)?redirectedfrom$=$MSDN

[42] Bsd make, "Bsd make," 2024. [Online]. Available: https://wiki.netbsd.org/tutorials/bsd_make/

[43] Gradle Build Tool, "Gradle build tool," 2023. [Online]. Available: https://gradle.org/

[44] Bazel Build, "Bazel build system," 2022. [Online]. Available: https://bazel.build/

[45] Java Programming, "Java programming language compiler," 2023. [Online]. Available: https://docs.oracle.com/javase//7/docs/technotes/tools/windows/javac.html

[46] Plugins, "Developing custom gradle plugins," 2023. [Online]. Available: https://docs.gradle.org/current/userguide/custom_plugins.html

[47] Gradle User Manual, "Gradle user manual," 2023. [Online]. Available: https://docs.gradle.org/current/userguide/userguide.html

[48] Bazel Manual, "Bazel manual," 2022. [Online]. Available: https://bazel.build/docs

[49] Algorithms, "Build system rules and algorithms," 2022. [Online]. Available: https://gittup.org/tup/build_system_rules_and_algorithms.pdf

[50] IBM, "IBM rational clearcase," 2022. [Online]. Available: https://www.ibm.com/products/rational-clearcase

[51] Fabicate, "Fabricate build tool," 2022. [Online]. Available: https://code.google.com/archive/p/fabricate/

[52] Memoize, "Memoize build framework," 2022. [Online]. Available: http://benno.id.au/blog/2008/06/06/memoize-build-framework

[53] S. Spall, N. Mitchell, and S. Tobin-Hochstadt, "Build scripts with perfect dependencies," *Proc. ACM Program. Lang*, vol. 4, no. OOPSLA, pp. 169:1–169:28, 2020. [Online]. Available: https://doi.org/10.1145/3428237

[54] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, "Automated localization for unreproducible builds," in *Proc. 40th Int. Conf. Software Eng. (ICSE)*, Gothenburg, Sweden, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds., May 27–Jun. 3, 2018, pp. 71–81. [Online]. Available: https://doi.org/10.1145/3180155.3180224

[55] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie, "Root cause localization for unreproducible builds via causality analysis over system call tracing," in *Proc. 34th IEEE/ACM Int. Conf. Automated Software Eng. (ASE)*, San Diego, CA, USA, Nov. 11–15, 2019, pp. 527–538. [Online]. Available: https://doi.org/10.1109/ASE.2019.00056

[56] Z. Ren, S. Sun, J. Xuan, X. Li, Z. Zhou, and H. Jiang, "Automated patching for unreproducible builds," in *Proc. 44th IEEE/ACM 44th Int. Conf. Software Eng. (ICSE)*, Pittsburgh, PA, USA, May 25–27, 2022, pp. 200–211. [Online]. Available: https://doi.org/10.1145/3510003.3510102

[57] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proc. 40th Int. Conf. Software Eng. (ICSE)*, Gothenburg, Sweden, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds., May 27–Jun. 3, 2018, pp. 1078–1089. [Online]. Available: https://doi.org/10.1145/3180155.3180181

[58] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: How far are we?" in *Proc. 28th ACM SIGSOFT Int. Symp. Software Test. Anal. (ISSTA)*, Beijing, China, D. Zhang and A. Møller, Eds., Jul. 15–19, 2019, pp. 43–54. [Online]. Available: https://doi.org/10.1145/3293882.3330578

[59] D. Tarlow et al., "Learning to fix build errors with graph2diff neural networks," in *Proc. (ICSE): 42nd Int. Conference on Software Engineering, Workshops*, Seoul, Republic of Korea, Jun. 27–Jul. 19, 2020, p. 19. [Online]. Available: https://doi.org/10.1145/3387940.3392181

[60] Distcc, "Distcc: A fast, free distributed c/c++ compiler," 2024. [Online]. Available: https://www.distcc.org/

[61] Cache, "Cache—A fast c/c++ compiler cache," 2024. [Online]. Available: https://ccache.dev/

[62] K. Gallaba, J. Ewart, Y. Junqueira, and S. McIntosh, "Accelerating continuous integration by caching environments and inferring dependencies," *IEEE Trans. Softw. Eng.*, vol. 48, no. 6, pp. 2040–2052, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3048335.

[63] J. Lyu, S. Li, H. Zhang, L. Yang, B. Liu, and M. Rigger, "Towards efficient build ordering for incremental builds with multiple configurations," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1494–1517, 2024. [Online]. Available: https://doi.org/10.1145/3660774

Jun Lyu received the Ph.D. degree from Nanjing University, China, supervised by Prof. He (Jason) Zhang. His research interests include improving software development efficiency and quality, particularly improving software build efficiency and quality in practice. His recent research interests are in software testing and AI4SE.

Shanshan Li received the B.S. degree in software engineering, the M.S. degree in computer science and technology from China University of Petroleum, and the Ph.D. degree in software engineering from Nanjing University. She is an Assistant Professor with the Software Institute, Nanjing University. Her research interests include microservices architecture, blockchain-oriented software engineering, and evidence-based software engineering. She won the Best/Distinguished Paper Awards from APSEC 2016 and the Most Influential Paper Awards from APSEC 2023. Currently, she is a Professional Member of China Computer Federation.

Bohan Liu is an Assistant Professor on the tenure-track with the Software Institute, Nanjing University. His research interests include software processes, empirical software engineering, and machine learning. The specific research focuses on software process simulation, software engineering for artificial intelligence, DevOps, continuous integration, and other aspects. He has presided over a Youth Fund Project of the National Natural Science Foundation of China, participated in three projects of the National Natural Science Foundation of China, and presided over a project funded by the CCF-Huawei Populus Euphratica Foundation. Relying on these projects, he has published more than 20 papers in top venues, such as ACM Computing Surveys (CSUR), IEEE Transactions on Services Computing (TSE), Journal of Systems and Software (JSS), Journal of Software: Evolution and Process (JSEP), International Conference on Software Engineering (ICSE), ACM SIGSOFT Symposium on the Foundation of Software Engineering (FSE), International Conference on Software Analysis, Evolution, and Reengineering (SANER), Evaluation and Assessment in Software Engineering (EASE), International Conference on Software and System Process (ICSSP).

He Zhang (Member, IEEE) is a Full Professor with Software Institute, the Director of the DevOps+ Research Laboratory, Nanjing University, China, and a Principal Scientist with CSIRO, Australia. He joined academia after many years working in software industry. He undertakes research in software engineering, in particular software process, software architecture, DevOps, software security, and empirical and evidence-based software engineering. He has published over 200 peer-reviewed papers in prestigious international conferences and journals.

Guoping Rong (Member, IEEE) received the B.Sc. degree in computer science and technology, the M.Sc. degree in software theory, and the Ph.D. degree in applied software engineering from Nanjing University. Currently, he is a Faculty Member with the Software Institute, Nanjing University and the Director of the Joint Laboratory of Nanjing University and Transwarp on data technology. His research area includes software process, DevOps, AIOps, and empirical methodology.

Chenxing Zhong received the B.S. degree in software engineering from Nankai University. Currently, she is working toward the Ph.D. degree in DevOps+ Research Laboratory, Nanjing University, China. She undertakes research in software engineering, in particular software architecture, and software maintenance and evolution.

Xiaodong Liu is currently working toward the Ph.D. degree with DovOps+ Research Laboratory, Nanjing University, China. His research interests include ensuring the security of microservice architecture systems and their exposed web APIs.