Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph

Gang Fan
The Hong Kong University of Science and Technology
Hong Kong, China
gfan@cse.ust.hk

Chengpeng Wang
The Hong Kong University of Science and Technology
Hong Kong, China
cwangch@cse.ust.hk

Rongxin Wu
Department of Cyber Space Security,
Xiamen University
Xiamen, China
wurongxin@xmu.edu.cn

Xiao Xiao
Sourcebrella Inc.
Shenzhen, China
xx@sbrella.com

Qingkai Shi
The Hong Kong University of Science and Technology
Hong Kong, China
qshiaa@cse.ust.hk

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

ABSTRACT

Modern software projects rely on build systems and build scripts to assemble executable artifacts correctly and efficiently. However, developing build scripts is error-prone. Dependency-related errors in build scripts, mainly including missing dependencies and redundant dependencies, are common in various kinds of software projects. These errors lead to build failures, incorrect build results or poor performance in incremental or parallel builds. To detect such errors, various techniques are proposed and suffer from low efficiency and high false positive problems, due to the deficiency of the underlying dependency graphs. In this work, we design a new dependency graph, the unified dependency graph (UDG), which leverages both static and dynamic information to uniformly encode the declared and actual dependencies between build targets and files. The construction of UDG facilitates the efficient and precise detection of dependency errors via simple graph traversals. We implement the proposed approach as a tool, VERIBUILD, and evaluate it on forty-two well-maintained open-source projects. The experimental results show that, without losing precision, VERIBUILD incurs 58.2× less overhead than the state-of-the-art approach. By the time of writing, 398 detected dependency issues have been confirmed by the developers.

CCS CONCEPTS

• **Theory of computation** → **Program verification**; • **Software and its engineering** → *Software defect analysis*; **Software maintenance tools**; • **Social and professional topics** → **Software maintenance**.

KEYWORDS

build maintenance, build tools, dependency verification

1 INTRODUCTION

In the development of modern software, nearly every large project resorts to build systems (e.g., GNU MAKE, GNU AUTOTOOLS, BAZEL and NINJA) and the corresponding build scripts to automatically transform the source code into the executable software. Build scripts play a critical role in the build process, as they define the rules on compiling source code, resolving the dependencies of build targets, and packaging the binaries. However, developing build scripts is non-trivial and error-prone. Bugs in builds scripts are quite common, in both commercial and open-source projects. For example, Hochstein and Jiao [15] found that 11%-47% of test failures are build-related in scientific software. Seo et al. [34] also observed similar build-failure ratios in C++ and Java projects of Google Inc.

Among various kinds of build errors, dependency-related errors are the most common (accounting for 52.68%-64.71%) [34], and can be summarized into two categories: missing dependencies and redundant dependencies. Both are caused by inconsistency between the *declared dependency* (i.e., the dependency declared in the build scripts [17][1]) and the *actual dependency* (i.e., the dependency needed in the build time [17][2]). A missing dependency happens when a given actual dependency is not declared, which leads to failures in the parallel build or an incorrect build results in an incremental build [22]. A redundant dependency happens when no actual dependency corresponds to a given declared dependency, leading to a poor performance in the parallel or incremental builds.

---

[1]A target X has a declared dependency on a target Y, iff rules in the build scripts declare the dependence from X to Y.
[2]A target X is actually dependent on a target Y, iff Y must be present, built and up-to-date in order for X to be built correctly.

Detecting dependency-related errors is tedious and time-consuming [32]. Various techniques have been proposed to tackle different types of dependency-related errors. We classify these techniques into three categories based on how of dependency graphs represent the build targets and their dependencies.

The first category constructs the *static dependency graph* via statically analyzing the build scripts and the executed shell commands [1, 36, 37]. Since the static dependency graph is unsound by including only the explicitly declared dependencies, it is infeasible to detect both the missing and redundant dependencies due to the lack of actual dependencies. Therefore, this category of techniques can only detect bad smells in build definitions (e.g., the cyclic dependency issue).

The second category constructs the *dynamic dependence graph* via monitoring and capturing the actual dependency in build processes [22]. Note that, using only the dynamic dependence graph, it is still infeasible to detect the missing and redundant dependencies, due to the lack of declared dependencies for comparison. Therefore, Licker and Rice [22] proposed a build fuzzing technique, MᴋCʜᴇᴄᴋ, to infer the declared dependencies in the build scripts by triggering incremental builds. Specifically, by updating an input file of a project and triggering the incremental build, only those targets which have direct or transitive declared dependencies on this input file will be rebuilt. Therefore, by updating every input file one at a time to trigger incremental builds, MᴋCʜᴇᴄᴋ can eventually infer all the declared dependencies in the build scripts. Not surprisingly, this approach is not efficient for a large project with many source files and complex dependencies. For instance, it takes MᴋCʜᴇᴄᴋ 51.6 hours to complete all the possible incremental builds to infer the declared dependencies for OpenCV, a project with 3,746 source files in total, in our experiment.

The third category [6] leverages both the static and dynamic dependency graphs and cross-references them to discover the inconsistencies, essentially following the definitions of missing dependencies and redundant dependencies. Despite its potential to improve the efficiency, the unsoundness of the conventional static dependency graph is its Achillie's heel. Specifically, the static dependency graph lacks the *implicitly declared dependencies*, which are the dependencies between targets and those output files of its prerequisites that are not specified as targets. Common root causes of this implicity include the incorrect or the insufficient output files specified in the build scripts (see Section 5.3.3) and temporary files used during a build. To resolve the unsoundness of the static dependency graph, a tool used in the state-of-the-art technique [6], MAKAO [2], tries to extract the implicitly declared dependencies by parsing the command lines and discovering the files with certain extensions (e.g., ".c"). However, this approach cannot fully resolve the unsoundness. Typical examples are those binary files generated during runtime (files without extension) [2]. Moreover, build scripts may include some commands whose syntax would not be known in advance, and thus it would be impossible to devise an omnipotent parser to extract input and output files from commands. This unsoundness issue would lead to a large number of false positives when detecting inconsistencies [6]. Besides, due to the different granularities in the static and dynamic dependency graphs adopted in the existing work [6], it is difficult to remedy

the unsoundness issue of the static information via the dynamic information.

Our idea is also to leverage both static and dynamic dependency information but in a novel way: instead of cross-referencing two graphs, we construct a unified dependency graph (UDG) such that, for each build target, we are able to, via simple graph traversals, both infer the implicitly declared dependencies and compute the actual dependencies happening at runtime. Any difference in the results of these two traversals signifies potential inconsistencies and generates bug reports. Note that, UDG is novel such that both the declared and actual dependencies are fused in an uniform representation. This makes it feasible to infer the implicitly declared dependencies of a target via monitoring file I/O operations of its prerequisite targets during an actual build. As such, we can guarantee the soundness of the declared dependencies. Building UDG is also non-trivial. We propose a novel build instrumentation approach (See Section 3.1.2) to constructing the actual dependencies with respect to each build target, and encode them with the static dependency graph.

Note that, the design of UDG is not restricted to a certain build system. To demonstrate its generality, we have constructed UDGs for different build systems including make-based systems (e.g., GNU Mᴀᴋᴇ and CMᴀᴋᴇ) and Nɪɴᴊᴀ.

To evaluate the effectiveness of our proposed technique, we implemented a tool, VᴇʀɪBᴜɪʟᴅ, and applied it to forty-two well maintained open-source projects. The experimental results demonstrated that VᴇʀɪBᴜɪʟᴅ has a good performance in efficiency and precision. In the forty-two mature open source projects, VᴇʀɪBᴜɪʟᴅ found 2,498 dependency issues with a low overhead. Of all the detected issues, 2,217 are classified as true issues. By the time of writing, 398 of 703 submitted issues had been confirmed by the original developers.

In this paper, we make the following contributions:

- We present the design and the implementation of VᴇʀɪBᴜɪʟᴅ, a novel approach to detecting inconsistencies in the build systems. VᴇʀɪBᴜɪʟᴅ is more precise and more powerful than the state-of-the-art approaches.
- We propose a new dependency representation, namely the unified dependency graph(UDG), which faithfully encodes the actual and the declared dependencies between input/output files and each target. Based on the UDG, we can find build dependency-related errors by graph traversal on the UDG.
- We select four typical dependency-related errors and implement four checkers in VᴇʀɪBᴜɪʟᴅ. The experimental results show that VᴇʀɪBᴜɪʟᴅ achieves a high precision with an acceptable time cost. Our approach is more scalable compared with the state-of-the-art techniques.

The paper is organized as follows. We first present some motivating examples in Section II. Section III illustrates our approach. The implementation details and the evaluation are presented in Section IV and Section V respectively. Section VI discusses some related works and Section VII concludes this paper.

```
1  # Makefile
2  check: test-file test-static test-dynamic
3    ./build/test-file
4    ./build/test-static
5    ./build/test-dynamic
6
7  test-file: test-file.c test-dynamic test-static
8    cc test-file.c mpc.c -lm -o build/test-file
9
10 test-static: test-static.c libmpc.a
11   cc test-static.c -lm -lmpc -static -o build/test-static
12
13 test-dynamic: test-dynamic.c libmpc.so
14   cc test-dynamic.c -lm -lmpc -o build/test-dynamic
```

Figure 1: An Example Makefile from *MPC*

2 PRELIMINARIES

2.1 Build Systems and Scripts

Modern software projects consist of a large body of source files, libraries and resources. To build or test a software, developers write build scripts (e.g., Makefiles) to instruct a build tool (e.g., GNU MAKE [10], NMAKE [31], BUILDCOP [9]) to execute build commands in the order specified in the scripts. Among all the build tools, GNU MAKE is one of the most widely used build systems [30] and has influenced the design of all its successors. However, GNU MAKE is more prone to errors than others, which has led to the invention of many tools (e.g., AUTOTOOLS [8], CMAKE [27]) that help generate Makefiles automatically. Our research and implementation focus on detecting issues in projects managed by GNU MAKE. However, the method we proposed in this paper can also be applied to other build systems. We refer to GNU MAKE as MAKE for simplicity.

Consider a running Makefile build script example shown in Figure 1 simplified from an open-source project MPC[3]. It contains four dependency rules for four targets: *check*, *test-file*, *test-static*, and *test-dynamic*. Each rule specifies the **target** (e.g., *test-file*), the **prerequisites** for building this target (e.g., *test.c*, *test-dynamic* and *test-static*) and the **recipe**, which is a series of shell commands to build this target (e.g., *"cc test.c mpc.c -lm -o build/test-file"*). Based on the rule, the target has the declared dependency on its prerequisite.

Most build tools, including GNU MAKE, BAZEL and NINJA, support advanced features such as the **incremental build** and the **parallel build** for accelerating the build process. In an incremental build, previous build results are reused and only a subset of targets are rebuilt. For example, if *test-file.c* in Figure 1 is modified, only *test-file* and *check* will be rebuilt. Other targets including *test-static*, *test-dynamic* will not be built. In a parallel build, the build system analyzes the build scripts first to decide which targets can be built simultaneously and distributes tasks to different CPU cores to achieve shorter wall-clock build time. For instance, commands for building the targets, *test-static* and *test-dynamic*, can be executed in parallel since they are independent of each other.

2.2 Missing and Redundant Dependencies

Build scripts are error-prone. Wrong dependencies can lead to broken incremental and parallel builds, or even to incorrect and unreproducible builds. Two most common dependency-related errors in build scripts are **Missing Dependency** and **Redundant Dependency** .

---

[3]https://github.com/orangeduck/mpc

```
1  #include "ptest.h"
2
3  void suite_core(void);
4  void suite_regex(void);
5  void suite_grammar(void);
6  void suite_combinators(void);
7
8  int main(int argc, char** argv) { ...
```

Figure 2: Content of test-file.c

2.2.1 Missing Dependency (MD). An MD is a dependency for building a target that is not specified in the script. MDs are the most common errors in build scripts. They could lead to an incorrect dependency graph being constructed and can corrupt an incremental build and parallel build.

In Figure 1, executing the recipe for *test-file* will read the source file *mpc.c*. However, *mpc.c* is not specified as a dependency of *test-file* in the build script. If *mpc.c* is modified, the build system will not rebuild *test-file* in an incremental build, which leads to incorrect build results. Another common case of MD is related to header files inclusion. Figure 2 shows the content of *test-file.c* where *ptest.h* is included as a header. The compilation of *test-file.c* implicitly depends on the file *ptest.h*. However, this implicit dependency is not specified in the build script.

2.2.2 Redundant Dependency (RD). An RD is a redundant prerequisite specified for a target which the build does not depend on. Similar to an MD, an RD can also mislead the construction of the dependence graph used in a build system. For example, the target *test-file* in Figure 1 has two RDs: *test-dynamic* and *test-static*. However, the build of *test-file* does not rely on any outputs of *test-dynamic* or *test-static*. RDs can cause unnecessary targets to be built in an incremental build. If *libmpc.so* is modified, the build system will mistakenly decide *test-file* should be rebuilt, which is clearly unnecessary. RDs could also reduce the parallelism of a parallel build. In this example, a build system should be able to build the three targets *test-file*, *test-static* and *test-dynamic* in parallel since they do not depend on each other. Due to the RDs, the build system will mistakenly decide that the target *test-file* should be built after the target *test-dynamic* and *test-static* are built, even when there are spare CPU cores.

2.3 Building Declared and Actual Dependencies

In VERIBUILD, we propose a new type of dependency graph, namely unified dependecy graph (UDG), which uniformly encodes the actual and the declared dependencies in terms of build targets and files. Figure 3 shows the UDG of the example build script in the Figure 1.

The subgraph which consists of the static dependency edges and the static spawn edges in Figure 3 is the conventional static dependency graph. Specifically, a static dependency edge represents the dependency between targets, and a static spawn edge represents the case in which a build target is defined as a file. As discussed in Section 1, the implicitly declared dependencies could be missed by statically parsing the build scripts. For example, the target *check* depends on the target *test-static* which could generate the output file *build/test-static*, and thus *check* has implicitly declared dependency on *build/test-static*. Although the shell script in *check* is a typical compiling command and not difficult to parse in this case, it still
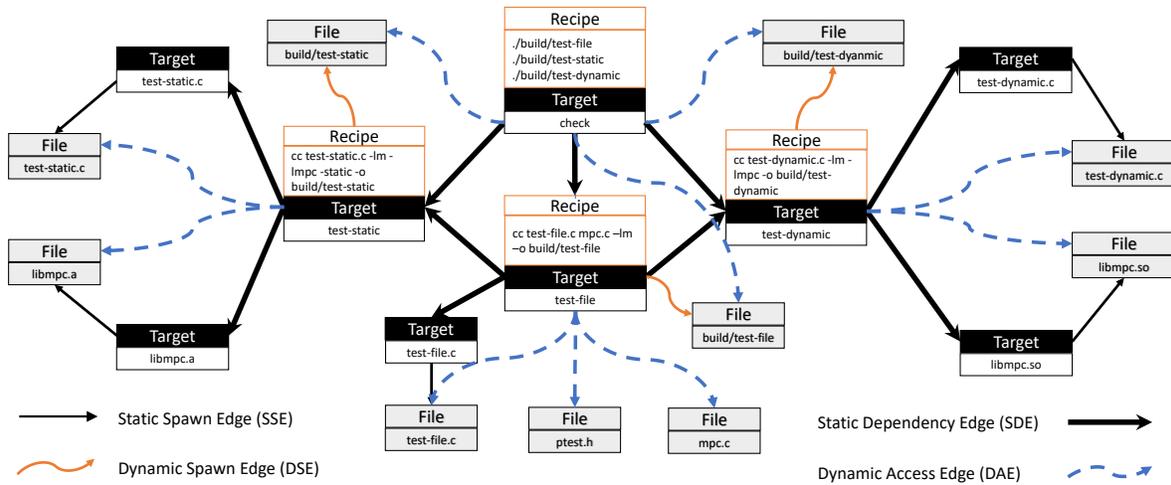
Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang

Figure 3: A UDG for the Example Build Script Shown in Figure 1

requires certain engineering effort to customize a parser for this command and is hard to generalize to arbitrary commands. To avoid the parsing trouble, we leverage the build instrumentation (See Section 3.1.2) to capture the output files of each build target (i.e., dynamic spawn edges) in the build time, which can guarantee the soundness of the implicitly declared dependencies. Therefore, a **sound static dependency graph** includes the static dependency edges, the static spawn edges, and the dynamic spawn edges in UDG.

The subgraph which consists of the dynamic access edges and the dynamic spawn edges represents the dynamic dependency graph in terms of build targets and files. Specifically, a dynamic access edge represents an actual dependency between a target and one of its input files, while a dynamic spawn edge represents an actual dependency between a target and one of its output files. Both types of edges are built via our proposed build instrumentation approach (See Section 3.1.2).

2.4 Detecting MDs and RDs

With the construction of UDG, we can readily convert the problem of detecting the two common dependency-related build errors, i.e., MDs and RDs, into the graph traversal problem.

Take Figure 3 as an example. To detect the MD issues for the build target *test-file* (missing the declared dependency on *ptest.h* and *mpc.c*), we first traverse from *test-file* along with the edges of the sound static dependency graph to discovers the set of files on which *test-file* has declared dependencies, including only *test-file.c*. Then, we traverse along with edges of the dynamic dependency graph to discover the set of files on which *test-file* has actual dependencies, including *test-file.c*, *ptest.h*, and *mpc.c*. From the inconsistency between the two sets of files, we can easily discover that *ptest.h* and *mpc.c* are the missing dependencies for *test-file*.

Detecting the RD issues for the build target *test-file* follows a similar process. First, we discover the actual dependency files for *test-file*, including *test-file.c*, *ptest.h*, and *mpc.c*. Then, to verify whether *test-static* is an redundant declared dependency for *test-file*, we traverse from *test-static* to discover the set of the files on which *test-static* has declared dependencies, including *test-static.c*

and *libmpc.a*, which do not correspond to the actual dependency files of *test-file*. Based on the definition of the RD issue, we confirm *test-static* is a redundant declared dependency for *test-file*.

The above running examples demonstrate the basic idea of using the UDG to detect the MD and the RD issues. In Section 3, we explain the technical details of the approach, as well as its extensibility to other dependency-related errors.

3 APPROACH

We propose a unified approach to analyzing build scripts, which analyzes the static dependency information specified in the build scripts together with the dynamic dependency information to obtain a sound profile of the dependency relations, encoded by a novel data structure, the UDG. The graphical representation of dependency information enables us to analyze build dependencies with graph traversals.

In this section, we first formally define the *UDG* (Section 3.1), followed with a unified construction method. The construction method consists of two steps: *Script Parsing*, parsing and evaluating the build scripts to infer the static dependency graph (Section 3.1.1), and *Build Instrumentation*, instrumenting the build process to infer actual dependencies (Section 3.1.2). Detecting MDs and RDs can be modeled as graph queries on the UDG (Section 3.2). In order to demonstrate the extensibility of our approach, we also discuss two analyses for detecting File-Target Inconsistencies and File Race Conditions (Section 3.3).

3.1 Unified Dependency Graph

We first define the Unified Dependency Graph (UDG) as follows:

Definition 1. Unified Dependency Graph (UDG) $UDG = (V, E)$, where V is the set of nodes and E is the set of edges.

$V = V_T \sqcup V_F$: V_T and V_F are two disjoint subsets, where

- $t \in V_T$ is a **target node** that represents a target in the build scripts. Each *target node* is bound with a recipe. A recipe is a list of commands to build the target.
- $f \in V_F$ is a **file node** that represents a file.

$E = E_{SD} \sqcup E_{SS} \sqcup E_{DA} \sqcup E_{DS}$: E consists of four disjoint subsets, each of which contains a specific kind of edges defined as follows.

- $e_{SE} \in E_{SE} \subseteq V_T \times V_T$ is a **Static Dependency (SD) edge** from a target node to another (e.g., all the thick solid straight edges in Figure 3), which represents the dependency relation between the two targets specified in the build scripts.
- $e_{SS} \in E_{SS} \subseteq V_T \times V_F$ is a **Static Spawn (SS) edge** from a target node to a file node (e.g., all the thin solid straight edges in Figure 3), which represents the output file for target is statically available before a build.
- $e_{DA} \in E_{DA} \subseteq V_T \times V_F$ is a **Dynamic Access (DA) edge** from a target node to a file node (e.g., all the dashed curved edges in Figure 3), which represents that the file is accessed when building the target.
- $e_{DS} \in E_{DS} \subseteq V_T \times V_F$ is a **Dynamic Spawn (DS) edge** from a target to a file node (e.g., all the solid curved edges in Figure 3), which means that building the target will output the file.

The DA edges represent the actual dependencies, and the SS/DA/DS edges represent the declared dependencies. Provided with a clean build, we can construct the UDG for a project with the following method consists of two phases: **Script Parsing** and **Build Instrumentation**.

3.1.1 Script Parsing. We parse and evaluate the build scripts to construct the static dependency graph of a UDG. Parsing user-written build scripts such as Makefiles can be difficult, due to the intensive usages of macros and variables, as well as explicit and implicit pattern rules embedded in build tools. Instead, we parse the build scripts after the build tool has evaluated them (e.g., after macros are expanded and implicit rules are applied). In which, all the rules are specified in the standard form with macros being expanded and evaluated, and all the pattern rules being applied as well. For each of the targets listed in the build scripts, we create one corresponding node. The commands for building a target are kept in the recipe of the target node. For each dependency rule specified in the build scripts, we create SD edges from the target node to all of its dependencies. For each target node with an empty recipe, we check whether its default output file (the file with the same name as the target name) exists in the file system. If the output file exists, we create an SS edge from the target node to a file node representing this file. After script parsing, all the target nodes, SD edges, and SS edges are constructed. File nodes representing the inputs of this build (e.g., source code files and configurations) have been created as well. The resulting graph faithfully describes the static dependencies specified in the build scripts.

3.1.2 Build Instrumentation. The purpose of build instrumentation is to obtain the actual input (DA edges) and output files (DS edges) for building a target. There are existing tools based on system call tracing (e.g., *strace* in Linux) and file change monitoring (e.g., *fswatch* in Linux) that can inspect actual file operations for a build. However, the resulting file operations are based on processes (*strace*) or paths (*fswatch*), and cannot easily be correlated to build targets. Thus, instead of monitoring file operations of a build, we perform a build instrumentation, which initiated the build of each target and captures file operations separately.

Based on the static dependency graph obtained in the script parsing phase, we follow the topological order of this graph to build all the targets by executing their recipes. We fork a new process for the execution of each recipe, and capture file-related system calls performed by the forked process and its sub-processes. For each file read operation, we create a DA edge from the target node to the file node. For each file write or create operation, we create a DS edge from the target node to the file node. After build instrumentation, we are able to get the complete UDG.

3.2 Dependency Analyses on the UDG

The UDG encodes both the actual dependencies (DA edges) and the declared dependencies (SD/SS/DS edges). It is non-trivial for existing approaches to obtain the files made available when building a target. With the UDG, this problem can be solved by traversing the graph from the target node along with declared dependency edges. All of the reachable files are available at the time of build. This uniform modeling of the declared and the actual dependencies eases the analyses of dependencies. In this subsection, we discuss how to detect MDs and RDs with the UDG. We also discuss two more analyses to demonstrate the broad applicability of the approach (Section 3.3).

3.2.1 Detection of Missing Dependencies(MDs). An MD occurs when a target depends on a file that is not declared in the build scripts (transitively). Detecting an MD for a target can be modeled as a graph query on the UDG:

Problem 1. *For target node t, if a file node f is reachable from t through the DA edges but not through the declared dependency edges (SD edges, SS edges, DS edges), f is an MD of t.*

To detect MDs for a target, we first analyze its declared dependencies. Algorithm 1 shows two procedures for querying available files for a target: AvFnBefore and AvFnAfter, for querying the files available before and after building a target. In this algorithm, $t.succs(edge = EdgeType(s))$ denotes querying t's direct successors reachable with edges of specific $EdgeType(s)$ on the UDG. The resulting file set for each target is cached for optimization purposes.

Algorithm 1 Available file nodes before and after building a target

---

procedure AvFnBefore(t: Target Node)
   $P_t := t.succs(edge = SD)$
   $S_t := t.succs(edge = SS)$
   $F_t := \bigcup_{p \in P_t} \text{AvFnAfter}(p)$
   return $S_t \cup F_t$
procedure AvFnAfter(t: Target Node)
   $S_t := \text{AvFnBefore}(t) \bigcup t.succs(edge = DS)$
   return S_t

Algorithm 2 shows how to detect all the MDs for a target, we compare the set of files accessed during its build (D_t) to the available file set (S_t). All files in $D_t \setminus S_t$ are MDs to this target.

3.2.2 Detection of Redundant Dependencies(RDs). An RD is a declared dependency of a target, which does not occur in the actual build time. Since the build scripts are written in a way where only

Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang

Algorithm 2 Missing Dependency Detection

Input: *UDG*, target node t
Output: the *MD* set of t
1: $S_t = \text{AvFnBefore}(t)$
2: $D_t = t.\text{succs}(edge = DA)$
3: $MD = D_t \setminus S_t$
4: **return** *MD*

---

one level of dependencies are specified. We only detect RDs in the direct dependencies of a target. The detection of RDs can be modeled as a graph problem on the UDG:

Problem 2. *If t_2 is a direct dependency of t_1, for every file node f that t_1 can reach through both declared dependency edges and actual dependency edges, if we can always find a declared dependency path to f that does not pass through t_2, then t_2 is an RD to t_1.*

To decide whether p is an RD for target t, we remove p from t's declared dependencies and check whether the declared dependencies in the remaining graph still cover the same actual dependencies when building t. Algorithm 3 shows the details of the detection. Since the available file nodes of different direct dependencies of a target may overlap, we use a map M at line 3 to record, for each file node, the number of direct dependencies it is available in. Line 9 to 12 perform the RD detection by checking whether removing a dependency may reduce the number of elements in M. The underlying problem of RD detection is set-cover [42], which is one of the NP-Complete problems. However, the size of a UDG is relatively small in practice. In the evaluation, the algorithm for detecting RDs always terminates in seconds.

Algorithm 3 Redundant Dependency Detection

Input: *UDG*, Target Node t
Output: the RD set of t: *RD*
1: $D_t := t.\text{succs}(edge = DA)$
2: $P_t := t.\text{succs}(edge = SD)$
3: $M := Map(FileNode \rightarrow Int)$
4: $RD := \emptyset$
5: **for** $p \in P_t$ **do**
6:     **for** $f \in \text{AvFnAfter}(p)$ **do**
7:         **if** $f \in D_t$ **then**
8:             $M[f] = M[f] + 1$
9: **for** $p \in P_t$ **do**
10:     **for** $f \in \text{AvFnAfter}(p)$ **do**
11:         **if** $f \in D_t$ and $M[f] == 1$ **then**
12:             $RD = RD \cup \{p\}$
13: **return** *RD*

3.3 More Analyses on the UDG

With the static and dynamic dependency information encoded in the UDG, we can perform various analyses for other dependency-related errors. We present two analyses for detecting **File-Target Inconsistencies** and **File Race Conditions**, respectively, in the build scripts.

3.3.1 Detection of File-Target Inconsistencies(FTIs). In most build systems, a *target* name does not have to be the same as its output file name. For example, many Makefiles have target names such as "all", "clean" and "test", however, they do not actually output such files. However, some build systems (e.g., Make) rely on the target name to be the output file name to compute dependencies for incremental build and parallel build. For example, in Figure 1, building target *test-file* will output a file "*build/test-file*" instead of "*test-file*" that is assumed by Make. When building "*test-file*" incrementally, Make will search for file "*test-file*" to decide whether the target *test-file* has been built. Since file "*test-file*" does not actually exist, Make will mistakenly decide that it has to be rebuilt. Moreover, the rebuild of *test-file* will cause all targets that depend on it to be rebuilt, which dooms the purpose of incremental build. We denote such an issue as a File-Target Inconsistency (FTI).

To detect FTIs, we compare the target names with their corresponding output file names. With the UDG, we could easily obtain all the actual output files for a target by traversing along the DS edges. With this method, we can detect that the output file name for target "*test-dynamic*" is "*build/test-dynamic*". We report it as an FTI issue and suggest the developers to use the filename directly to achieve a better performance.

3.3.2 Detection of File Race Conditions(FRCs). Two processes in a parallel build may access the same file for building different targets. If one of the two accesses is a write, a File Race Condition (FRC) might occur. An FRC may cause a parallel build to fail, damage the build artifacts, and results in flaky builds [20].

To detect FRCs for a file, we first identify all the targets that access it and then apply a May-Happen-in-Parallel(MHP) analysis [3, 28] to decide whether those targets may execute in parallel. For a file node, we first check whether it could be accessed by more than one target node. We collect all its predecessors through the DS edges and all predecessors through the DA edges. For any two targets that access this file node, if one is a write access, we check whether there is a path which consists of only SD edges from one to another. If there are no such edges, no order is specified for those two targets in the build scripts, and thus, they may be built in parallel, which could trigger the FRC.

4 IMPLEMENTATION

We implement the approach as a tool, VeriBuild, for analyzing projects maintained with Make or Ninja. We also implement the algorithms for detecting four kinds of dependency issues as "checkers" of VeriBuild. Figure 4 shows the architecture of VeriBuild. We first infer a dependency graph of all build targets by parsing the build scripts. Second, we instrument a customized builder to build each target according to the static dependency graph in the topological sorting order. When executing the recipe of a target, we trace the system calls to collect the files which are read, written, created, or deleted. After the UDG is constructed, we run the algorithms described in Section 3 to detect dependency issues.

For inferring the static dependency information, we customize a version of Make to export its dependency information together with the recipe information, and use the exporting feature supported by Ninja. This exporting feature is commonly supported
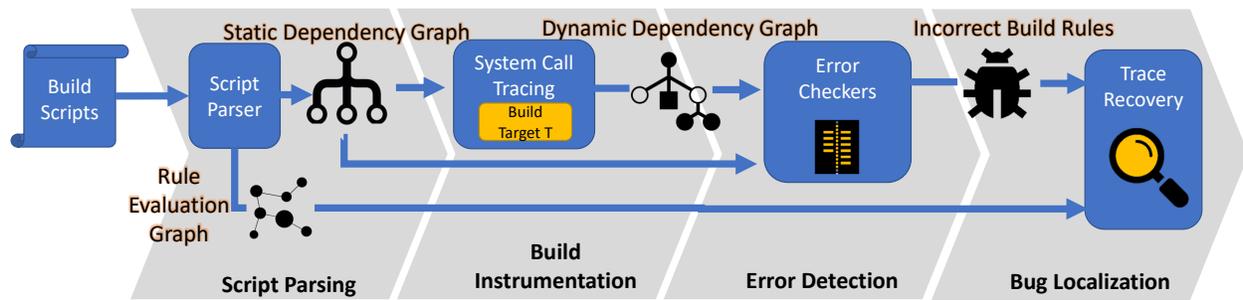
Figure 4: Architecture of VeriBuild

by many build tools. For example, Make can output its build profile with the option *-profile*, SCons has a similar option *−tree=all* and Bazel even provides a query interface for retrieving such build information [19]. Similar to MkFault [4], the customized Make also exports the evaluation graph to records the evaluation traces for variables. The evaluation trace is used for assisting developers in localizing the build rules for fixing the detected dependency issues. The discussion of bug localization is beyond the scope of this paper.

To construct the DS/DA edges for a UDG, we need to inspect file operations when executing a recipe. We use *ptrace*, a system call in Unix and Unix-like operating systems that allow one process to inspect and manipulate the internal state of another process. In order to correctly map the dynamic system call traces to corresponding targets, we execute the recipes of targets as described in Section 3.1.2. We fork a new process for executing each recipe and monitor the file-related system calls invoked by the forked process and all of its sub-processes. To be more specific, we capture and inspect system calls such as *read*, *write*, *open*, *unlink*, *execve* and so on.

Using *ptrace* incurs overhead for the processes being traced. Our tool slows down the build by a factor of two. Since we only build the project once for analyzing it, we believe this slowdown is acceptable in practice. An alternative approach would be hooking into the dynamic loading system (e.g., similar to the *ltrace* tool in Linux [21]), which can inspect library calls of dynamic linked libraries and binaries with less overhead. However, this method does not work with the static-linked libraries and binaries.

Note that, our technique has two requirements for build systems. First, the build scripts follow the "target-prerequisite-recipe" model to define the dependencies among build targets. Second, the build systems invoke separate processes for build tasks. Therefore, our technique can be used in the many build systems, including Make, Ninja, SCons, and Bazel. Our technique could not be applied to Java build systems such as Maven and Ant since they do not satisfy the above requirements. Specifically, Maven and Ant invoke the compiler as a library function call instead of creating a separate process. This makes us unable to extract the actual dependencies by inspecting system calls.

5 EVALUATION

5.1 Experimental Setup

We test VeriBuild on forty-two benchmark projects. We compare our approach to MkCheck [22], since it is the state-of-the-art technique for incorrect build rule detection. To evaluate the performance of VeriBuild, we design the following research questions:

- **RQ1:** How effectively does VeriBuild detect MDs and RDs, compared with MkCheck?
- **RQ2:** How efficiently does VeriBuild detect MDs and RDs, compared with MkCheck?
- **RQ3:** How well does VeriBuild perform in detecting other build issues that MkCheck cannot support?

For RQ1 and RQ2, we run both VeriBuild and MkCheck on forty-two benchmark projects. For each test, we enable only the checkers for detecting MDs and RDs that both tools support. In order to evaluate the effectiveness and practicability of our approach, we manually inspect and classify the results and also seek confirmation from the developers. In order to evaluate the efficiency of our approach, we compare the overall time cost and the overhead of both tools. For RQ3, we perform the detection of FTIs and FRCs on all the benchmark projects to demonstrate the extensibility of our approach. We manually inspect the reports and perform additional tests to check their validity.

All the experiments are performed on a computer running Ubuntu-16.04 with an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz forty-core processor, and 256GB physical memory.

5.2 Subjects

We set up a benchmark set consisting of forty-two projects that are maintained with Make (28), CMake (12), or Ninja(2), including well-known open-source projects as well as several personal projects evaluated in [22]. Table 1 shows the basic information of these projects. The number of files in those projects ranges from three to over forty thousand. Most of the subjects, such as Python, LLVM, and Redis, are well-maintained in terms of build scripts. For better presentation, we divide the subjects into three categories based on the number of source files: small-sized projects that include less than 100 source files, medium-sized projects that include 100~1,000 files, and large-sized projects that include more than 1,000 files.

5.3 Results

5.3.1 Effectiveness. To evaluate the effectiveness of our approach, we classify each bug report into two categories: *True Positive* (TP) and *False Positive* (FP). In terms of FPs, we refer to those reports that are unlikely to be fixed by developers. They are due

Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang

to developers' intention, dynamic features in the build scripts, or abnormal usages of the build system. To mark a report as a TP or an FP, we employ a three-phased procedure. First, we submit the report to its developers and mark the report as a TP or an FP once the developers confirm or deny it. Second, for an MD report that has not received a response from developers and for the projects with an overwhelming number of reports, we apply an automatic validator to reproduce the reported issue. We mark it as a TP if it is successfully reproduced and an FP otherwise. Third, for RD reports that can not be automatically validated, we manually label it accordingly. As it would be subjective in the last phase, to mitigate this issue, two authors of this paper individually perform checks on the results, and we only consider those reports as TPs if the two authors can reach a consensus. We also present the confirmation results in addition to the TPs and RDs in Table 1 for further inspections.

Moreover, we compare our approach with MkCheck, the state-of-the-art approach [22]. Note that, MkCheck and VeriBuild report MDs and RDs using different dependency relations. MkCheck describes MD or RD issues by using the dependencies between input/output files, while VeriBuild describes them by using the dependencies between targets. To ease the presentation, we denote an MD/RD edge as a missing/redundant dependency edge between targets. To ensure a fair comparison, we normalize the reports generated by MkCheck and VeriBuild using the same representation using MD/RD edges. For MkCheck, we try the best to map each file in a bug report into its corresponding target and then collect all the edges between the discovered targets. For VeriBuild, although we aggregate multiple MD/RD edges starting from the same target into one bug report, we can still recover their original MD/RD edges. In this way, we are able to compare the performance of two tools. Of all the forty-two projects, MkCheck is able to complete analyzing of 38 projects. Thus, we compare the two approaches with these projects only.

Table 1 shows the results: VeriBuild detects 2,051 MDs and 437 RDs in forty-two subjects, of which 2,217 reports are classified as TPs. VeriBuild manages to detect issues in most of those subjects (39/42). To further evaluate its effectiveness, we seek confirmation from the original developers of those projects. However, due to the heavy workload of reporting and communicating, we only submit reports for those projects with less than one hundred reports (except for Python, developers of which are responsive and help us confirm more than one hundred reports). Moreover, to avoid flooding developers with too many bug reports, we manually pre-screen the bug reports and only submit the severe ones. In total, we submit 703 reports to their developers. By the time of writing, 398 of them are confirmed by the developers, and the rest are unconfirmed.

Compare to MkCheck, VeriBuild surpasses it in terms of the number of the MD edges and the RD edges in 25 and 22 projects, respectively. The reason why VeriBuild can detect more RD/MD issues in these projects is that MkCheck only fuzzes files with some certain extensions and would inevitably miss some true cases for those files with unexpected extensions. We observed that MkCheck reports more MD edges in eight projects, and more RD edges in two projects, compared with VeriBuild. We investigated these reports and found that most of them are redundant. For example, if an MD edge $T_1 \rightarrow T_2$ (T_1 depends on T_2) is discovered by MkCheck,

Table 1: Classification of Detected Dependency Issues

Project	# MDs TP/ALL	# RDs TP/ALL	# Total TP/ALL	# Confirm AC/SUMB	# MD Edges VB/MC/=	# RD Edges VB/MC/=
LLVM	29/89	0/67	29/156	0/0	n/a	n/a
OpenCV	40/68	0/16	40/84	0/40	n/a	n/a
Python	153/158	146/153	299/311	138/138	4419/706/679	5406/5399/5079
PHP	383/383	0/34	383/417	0/0	22694/1/0	78/1/1
GMP	512/519	1/1	513/520	0/0	631/329/0	1/0/0
OpenSSL	346/347	0/1	346/348	0/0	10865/0/0	222/0/0
Capstone	47/47	51/58	98/105	98/98	198/5659/138	621/523/523
httpd	1/1	0/0	1/1	0/1	1/2/0	0/122/0
Redis	4/11	4/4	8/15	0/4	160/0/0	9/0/0
Bash	63/65	34/51	97/116	48/48	513/1482/278	378/142/45
Cppcheck	39/41	13/13	52/54	36/36	53/0/0	42/18/18
ck	12/12	1/1	13/13	0/13	181/180/180	3/0/0
tig	0/0	0/0	0/0	0/0	0/0/0	2/0/0
lighttpd	0/1	0/0	0/1	0/0	104/69785/0	0/0/0
Tmux	2/2	0/0	2/2	0/2	2/0/0	0/0/0
neven	85/85	0/0	85/85	0/85	1453/1459/1453	0/0/0
GNU Aspell	0/0	1/1	1/1	1/1	0/12/0	1/0/0
LAME	0/0	1/1	1/1	0/1	0/3/0	2/0/0
cJSON	1/1	0/0	1/1	0/1	159/0/0	58/0/0
zlib	5/6	0/0	5/6	0/5	96/2/2	0/0/0
lec	2/2	0/0	2/2	0/2	2/0/0	0/0/0
clib	44/44	0/3	44/47	44/44	178/147/147	6/6/6
8cc	16/16	1/1	17/17	0/17	16/0/0	1/0/0
fastText	11/11	4/4	15/15	0/15	11/0/0	4/0/0
gravity	23/23	0/0	23/23	0/23	262/0/0	0/0/0
x86-thing	17/17	0/0	17/17	0/17	76/76/76	0/0/0
grbl	16/16	0/0	16/16	0/16	20/20/20	0/0/0
Bftpd	13/14	12/14	25/28	25/25	14/0/0	111/97/97
gwion-util	1/1	0/0	1/1	1/1	18/0/0	10/0/0
cctz	0/0	0/0	0/0	0/0	0/0/0	0/0/0
fzy	6/6	0/0	6/6	0/6	17/17/17	0/0/0
libco	20/20	0/0	20/20	0/20	32/63/23	0/0/0
mpc	13/13	12/13	25/26	0/12	18/0/0	14/0/0
namespaced_parse	3/3	0/0	3/3	0/3	10/4/3	7/0/0
kleaver	7/7	0/0	7/7	0/7	11/2/0	0/0/0
agt-proof-of-stake	4/4	0/0	4/4	0/4	n/a	n/a
greatest	7/8	0/0	7/8	7/7	8/1/1	1/0/0
Stack-RNN	2/2	0/0	2/2	0/2	14/0/0	0/0/0
CacheSimulator	2/2	0/0	2/2	0/2	3/2/2	0/0/0
conceptnet-5.5-paper	1/1	0/0	1/1	0/1	n/a	n/a
http-parser	4/4	1/1	5/5	0/5	4/0/0	1/0/0
Generic-C-Project	1/1	0/0	1/1	0/1	4/1/1	2/0/0
Total	**1935/2051**	**282/437**	**2217/2488**	**398/703**		

TP = True Positive AC = Accepted SUMB=Submitted n/a = not available VB = Veribuild MC=MkCheck

all of the targets which depend on T_1 will be reported as missing dependency on T_2 correspondingly, which lead to a significant larger number of redundant reports. In summary, other than the implementation issues, VeriBuild and MkCheck achieve similar performance, in terms of the effectiveness. To further understand the effectiveness of VeriBuild, we investigate some true and false MD/RD cases.

True MDs. Many MDs are cases of missing source files. They have various causes such as unsynchronized build scripts and source files, misuses of the implicit rules in Makefile, and clerical errors. Figure 6 shows an MD report we generated for *agt-proof-of-stake*[4], which uses Ninja to maintain its build. Two prerequisites of target *report.pdf* are missing: *cite.bib* and *llncs2e/llncs.cls*. The incremental build of *report.pdf* will be incorrect if any one of these two files is changed. Many other MD reports are the cases in which C/C++ header files are missing. Specifically, in order to decide the right files that a compilation may access, the headers directly and indirectly included in the source file should be carefully considered for the declared dependencies. Figure 5 shows an MD we reported in *fastText*[5], a library for text classification and representation learning from Facebook. Line 47 in Makefile specifies the dependencies of *model.o*. However, it misses the header file *utils.h* included in *model.cc* and all the headers included in *model.h*. One way to prevent these problems is to use the Auto-Dependency Generation [25]

---

[4]https://github.com/lucaspena/agt-proof-of-stake
[5]https://github.com/facebookresearch/fastText

```
46   # Makefile
47   model.o: src/model.cc src/model.h src/args.h
48   $(CXX) $(CXXFLAGS) -c src/model.cc
49
```

```
8    /* model.cc */          8    /* model.h */
9    #include "model.h"       9    #pragma once
10   #include "loss.h"        10
11   #include "utils.h"       11   #include <memory>
12                            12   #include <random>
13   #include <assert.h>      13   #include <utility>
14   #include <algorithm>     14   #include <vector>
15   #include <stdexcept>     15
16                            16   #include "matrix.h"
17   namespace fasttext {     17   #include "real.h"
18                            18   #include "utils.h"
19   Model::State::State(...) 19   #include "vector.h"
20   : lossValue_(0.0),       20
21   nexamples_(0),           21   namespace fasttext {
22                            22
```

Figure 5: An Example MD from *fastText*

```
28   # build.ninja
29   build stage-1.out : maude stage-1.maude
30
31   build report.pdf : pandoc report.md | template.latex
32   flags = --template template.latex --pdf-engine=xelatex $
33   --filter pandoc-citeproc --bibliography=cite.bib
34
```

Figure 6: An Example MD from *agt-proof-of-stake*

mechanism provided by some compilers, and another way is to use VERIBUILD to periodically check the build scripts.

False MDs. False MDs are mainly caused by non-deterministic features in Make. For example, Make supports the use of conditional branches in a recipe. The execution of a recipe may take different branches across different builds, and all are considered valid. However, our approach assumes that every file access leads to unconditional dependency, and thus reports false positive in such cases. Figure 8 shows such an example: at line 947, the recipe will continue no matter the folder *tests* exists or not (it will be created in this case). One possible solution to reduce those false MDs is to do heavier static analysis on the build scripts and try to enumerate all its possible behaviors.

True RDs. In order to fix missing dependencies, developers may specify too many extra dependencies. Since correctly handling the C/C++ header dependencies is difficult, many projects specify all headers as the dependencies for all objects. Figure 7 shows such an example extracted from *Python*. It specifies all headers (*PYTHON_HEADERS*) as the dependencies of all objects (*LIBRARY_OBJS* and *MODOBJS*), which results in hundreds of RDs being reported (Table 1). Similar to the MDs, the header problems could also be solved with Auto-Dependency Generation or a tool such as VERIBUILD. Other RDs are just the simple mistakes made by developers, such as the RD in Figure 1.

False RDs. When a dependency is identified as an RD of a target by VERIBUILD, it means the build of that target indeed does access any file that exclusively provided by this dependency. However, some of them are unlikely to be fixed by developers due to various reasons. For these RDs, we mark them as False RDs in Table 1. For example, Figure 8 shows the content of *Makefile.in* in *Bash*. VERIBUILD reports *Makefile* and *config.h* are two RDs of the target *bashbug*, since they are not accessed by any process

```
431  # Makefile.pre.in
432  LIBRARY_OBJS= \
433  $(LIBRARY_OBJS_OMIT_FROZEN) \
434  Python/frozen.o
435
```

```
962  PYTHON_HEADERS= \
963  $(srcdir)/Include/Python.h \
964  $(srcdir)/Include/abstract.h \
965  $(srcdir)/Include/asdl.h \
966  $(srcdir)/Include/ast.h \
967  ...
968
```

```
1095 $(LIBRARY_OBJS) $(MODOBJS) Programs/python.o: $(
        PYTHON_HEADERS)
1096
```

Figure 7: An Example RD from *Python*

```
594  # Makefile.in
595  bashbug: $(SDIR)/bashbug.sh config.h Makefile $(VERSPROG)
596  @sed -e "s%!PATCHLEVEL!%$(PatchLevel)%" \
597  $(SDIR)/bashbug.sh > $@
598  @chmod a+rx bashbug
599
```

```
945  # Makefile.in
946  test tests check: force $(Program) $(TESTS_SUPPORT)
947  @-test -d tests || mkdir tests
948  @cp $(TESTS_SUPPORT) tests
949  @( cd $(srcdir)/tests && \
950  PATH=$(BUILD_DIR)/tests:$$PATH THIS_SH=$(THIS_SH) $(SHELL) $
        {TESTSCRIPT} )
951
```

Figure 8: An Example RD from *Bash*

when building the target *bashbug*. However, they should not be removed from the rule because it is the developers' intention to rebuild *bashbug* if these two files are modified. The file *config.h* is generated automatically when the project is re-configured, and its change is a flag to tell the whole project to rebuild.

Another category of false RDs are order-only prerequisites. Developers specify the build rules to enforce the build order of different targets, instead of the input-output dependencies. For example, in Figure 8, at line 946 in *Makefile.in*, the developer specifies *force* as the dependency of *test*, *tests* and *check*. The build of the three targets does not rely on any output from building *force*, but the developer intends to build them only after the build of *force*. To be more efficient, one can specify such dependencies as order-only prerequisites [26] supported by Make, and VERIBUILD can utilize this information to filter out false RDs caused by order-only prerequisites.

Answer to RQ1 VERIBUILD manages to detect 2,051 MDs and 437 RDs in forty-two real open-source projects, of which 2,217 are true bugs reports and 398 reports have been confirmed by the developers, which demonstrates its effectiveness.

5.3.2 Efficiency. To evaluate the efficiency, we compare VERIBUILD with MKCHECK, in terms of the time cost. Although the prior study [22] does not report the time cost of MKCHECK, we directly run their tool on all the subjects in our experiment and record the execution time. MKCHECK requires that a target name being specified for constructing the dependency graph. Thus, we run both tools for only the default targets in the build scripts.

Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang

Table 2: Performance Data for VeriBuild and MkCheck

Origin	Project	# Lines	# Files	Clean Build (sec)	MkCheck(sec) Dep Graph	Build Fuzz	Race Detection	Overhead	Total	VeriBuild(sec) Script Parsing	Build INSTR	MD/RD Detection	Overhead	Total	Speedup Overhead	Total
(#Files < 100) Open Source Projects	Generic-C-Project	15	3	0.11	0.25	6.47	6.37	118.0X	13.09	0.12	0.46	0.03	4.5X	0.61	26.1X	21.6X
	http-parser	6,347	6	33.09	33.84	138.86	167.77	9.3X	340.47	0.04	35.10	0.03	0.1X	35.17	147.5X	9.7X
	conceptnet-5.5-paper	3,041	6	1.26	n/a	n/a	n/a	n/a	n/a	0.12	1.68	0.03	0.5X	1.83	n/a	n/a
	CacheSimulator	953	10	0.32	0.71	15.61	2.68	58.4X	19.00	0.04	0.92	0.02	2.1X	0.98	28.3X	19.4X
	Stack-RNN	1,746	10	15.53	8.18	191.22	15.68	12.8X	215.08	0.02	16.02	0.06	0.0X	16.11	346.5X	13.4X
	greatest	1,581	11	0.61	1.21	16.70	11.09	46.5X	29.00	0.03	1.59	0.03	1.7X	1.66	27.2X	17.5X
	agt-proof-of-stake	3,687	12	14.81	n/a	n/a	n/a	n/a	n/a	0.15	15.70	0.05	0.1X	15.90	n/a	n/a
	kleaver	734	14	0.72	1.72	35.86	0.97	52.5X	38.55	0.21	2.38	0.04	2.6X	2.63	19.8X	14.7X
	namespaced_parser	599	15	0.62	0.53	8.89	14.96	38.3X	24.38	0.04	0.69	0.02	0.2X	0.75	188.0X	32.7X
	mpc	4,566	16	21.22	24.83	444.39	91.91	25.4X	561.13	0.03	26.79	0.07	0.3X	26.89	95.2X	20.9X
	fzy	3,798	30	0.81	1.85	28.96	16.41	57.3X	47.22	0.03	1.87	0.04	1.4X	1.95	40.8X	24.3X
	libco	3,727	31	3.44	7.06	52.49	34.05	26.2X	93.60	0.05	8.05	0.15	1.4X	8.25	18.7X	11.3X
	cctz	7,782	33	14.83	21.46	184.58	64.81	17.3X	270.85	0.10	23.66	0.24	0.6X	24.00	27.9X	11.3X
	gwion-util	1,238	34	0.67	1.65	64.68	19.69	127.4X	86.02	0.09	2.10	0.07	2.4X	2.26	53.7X	38.1X
	Bftpd	5,003	37	1.91	3.17	90.20	26.42	61.7X	119.79	0.03	4.06	0.09	1.2X	4.18	51.9X	28.7X
	grbl	4,514	51	1.21	2.63	108.09	38.11	122.0X	148.83	0.64	3.38	0.09	2.4X	4.11	50.9X	36.2X
	x86-thing	2,903	55	0.08	0.24	99.77	56.03	1,949.5X	156.04	0.08	0.11	0.02	1.7X	0.21	1,171.4X	732.1X
	gravity	17,817	56	9.60	13.27	275.97	58.82	35.3X	348.06	0.13	15.48	0.17	0.6X	15.77	54.8X	22.1X
	fastText	6,749	68	18.74	23.16	249.43	70.15	17.3X	342.74	0.19	26.18	0.21	0.4X	26.59	41.3X	12.9X
	8cc	10,165	82	1.21	2.38	126.71	31.91	132.1X	161.00	0.04	3.32	0.25	2.0X	3.61	66.6X	44.6X
(100 < # Files < 1000) Open Source Projects	clib	14,843	105	2.24	5.38	204.99	102.77	138.8X	313.14	0.08	7.27	0.19	2.4X	7.54	58.7X	41.5X
	lec	4,397	106	14.41	24.66	192.59	19.52	15.4X	236.77	0.47	28.67	0.03	1.0X	29.17	15.1X	8.1X
	zlib	28,998	106	7.27	10.23	110.52	84.31	27.2X	205.06	0.07	12.03	0.19	0.7X	12.29	39.4X	16.7X
	cJSON	19,437	107	0.51	0.38	11.89	8.75	40.2X	21.02	0.08	1.15	0.03	1.5X	1.26	27.3X	16.7X
	LAME	47,124	146	8.60	18.70	429.36	224.72	77.2X	672.78	0.81	28.23	0.44	2.4X	29.48	31.8X	22.8X
	GNU Aspell	30,821	184	45.04	66.72	2,760.43	532.27	73.6X	3,359.42	0.97	80.96	1.13	0.8X	83.05	87.2X	40.5X
	neven	22,743	186	7.92	10.79	377.87	182.07	71.1X	570.73	0.15	12.66	0.44	0.7X	13.24	105.7X	43.1X
	Tmux	52,244	206	24.71	44.71	678.33	333.32	41.8X	1,056.36	0.44	63.38	1.70	1.7X	65.52	25.3X	16.1X
	lighttpd	61,072	232	33.09	61.90	1,440.96	483.14	59.0X	1,986.00	3.12	83.29	2.13	1.7X	88.55	35.2X	22.4X
	tig	43,394	238	8.42	16.34	752.33	103.89	102.6X	872.56	0.18	20.46	0.54	1.5X	21.17	67.8X	41.2X
	ck	31,743	244	1.53	2.84	480.65	28.60	333.7X	512.09	0.40	3.36	0.03	1.5X	3.79	225.4X	134.9X
	Cppcheck	182,121	399	148.61	182.06	3,614.48	437.18	27.5X	4,233.72	0.09	209.75	1.14	0.4X	210.98	65.5X	20.1X
	Bash	126,820	442	41.53	63.57	2,100.20	485.56	62.8X	2,649.33	0.48	89.56	1.32	1.2X	91.36	52.3X	29.0X
	Redis	111,435	457	20.10	27.92	1,440.26	152.70	79.6X	1,620.88	0.43	40.12	0.09	1.0X	40.64	77.9X	39.9X
	httpd	199,327	502	113.54	243.03	1,388.86	1,562.61	27.1X	3,194.50	2.26	398.03	0.36	2.5X	400.66	10.7X	8.0X
	Capstone	161,930	604	36.92	51.51	1,734.99	296.26	55.4X	2,082.76	0.44	71.06	0.49	1.0X	71.98	58.3X	28.9X
(# Files > 1000) Open Source Projects	OpenSSL	580,653	1,767	290.03	n/a	n/a	n/a	n/a	n/a	3.68	340.06	18.81	0.3X	362.54	n/a	n/a
	GMP	258,875	1,878	87.35	179.13	4,057.00	6,159.08	118.0X	10,395.21	2.72	220.76	10.82	1.7X	234.30	70.1X	44.4X
	PHP	766,944	2,077	521.51	764.33	6,293.62	1,615.85	15.6X	8,673.80	5.78	812.63	7.42	0.6X	825.82	26.8X	10.5X
	Python	1,053,963	2,645	161.87	197.48	29,432.70	1,077.04	188.7X	30,707.22	0.88	264.71	3.26	0.7X	268.84	285.5X	114.2X
	OpenCV	1,670,991	3,746	2,694.86	3,140.96	185,822.75	4,193.23	70.7X	193,156.94	4.13	3744.65	45.54	0.4X	3,794.32	173.2X	50.9X
	LLVM	5,670,236	41,788	19,113.58	n/a	n/a	n/a	n/a	n/a	9.77	21,576.22	435.45	0.2X	22,021.44	n/a	n/a
	Sum:			23,508.35	>5,260.78	>245,463.66	>18,810.70	56.2X	>269,535.13	39.37	28,281.17	533.16	0.8X	28,853.69		
	%				2.0%	91.1%	7.0%	%		0.1%	98.0%	1.8%	Average:		58.2X	26.5X

n/a = data unavailable due to the failure of MkCheck.

Table 2 shows the evaluation results. Overall, VeriBuild is one or two orders of magnitude faster than MkCheck for almost all subjects. Especially for the large subjects, VeriBuild achieves speedups from 10×(PHP) to 114×(Python). The time cost of MkCheck in checking large projects such as OpenCV (193,157s) is high, which hinders its wide adoption. Note that, VeriBuild achieves such a high performance but does not sacrifice the effectiveness (see Section 5.3.1).

To understand the performance bottleneck of each tool, we further investigate the time cost of each step for both tools. MkCheck includes three steps, i.e., Dependency Graph Construction, Build Fuzz, and Race Detection. VeriBuild includes three steps, i.e., Script Parsing, Build Instrumentation, and MD/RD Detection. The results show that both tools incur some overhead for inspecting the system calls for a build (MkCheck-build v.s. build, VeriBuild-build v.s. build) and the overheads are in the same level (MkCheck 59%, VeriBuild 108%), since both tools use the *ptrace* system call to monitor files operations of a clean build. VeriBuild incurs slightly higher overhead since it also parses and analyzes the build scripts in this stage.

The testing time of MkCheck is dominated by the build fuzzing time (91.1%). As we have discussed in Section 2, the primary purpose of build fuzzing is to obtain static dependency information. As a comparison, VeriBuild parses and evaluates build scripts to obtain the static dependency information, and it costs only 39.37 seconds in total, which is 0.016% (39.37/245,463.66) of the build fuzzing time of MkCheck. The analysis time cost is relatively trivial for VeriBuild.

For analyzing all projects, VeriBuild spends only 533.16 seconds in total for analyzing both MDs and RDs.

Answer to RQ2 In this experiment, VeriBuild is 26.5X faster than MkCheck, on average. Moreover, VeriBuild only incurs <90% runtime overhead compared with the native clean build (i.e., without instrumentation) on average.

5.3.3 Extensibility. To demonstrate that our approach has better extensibility than the fuzzing testing for detecting build-related issues, we have implemented two checkers to check FTIs and FRCs. With the UDG being constructed, the effort for implementing the two checkers are rather trivial. Both checkers are written with less than 60 lines of Python code and have taken less than one hour for one of our authors for the implementation, which demonstrates great extensibility of applying our approach to new build-related problems. We run the two checkers on the forty-two benchmarks we used in RQ1 and RQ2. In total, we detected six FTIs in two projects and eight FRCs in one project. The detailed results can be found on the project's website. Here we only discuss two representative cases.

File-Target Inconsistency(FTI). Figure 1 shows a FTI we detect in *MPC*. The target names "*test-file*", "*test-static*" and "*test-dynamic*" are inconsistent with the files these targets actually output. Due to these three FTIs, the build system will mistakenly rebuild these three targets even for an incremental build. After correcting these FTIs in the Makefile, the build time for the incremental build is reduced from 21 seconds to 0.03 seconds.

```
41  # Makefile
42  PROGS = colib example_poll example_echosvr example_echocli
        ...
43
44  all:$(PROGS)
45
46  libcolib.a: $(COLIB_OBJS)
47  $(ARSTATICLIB)
48  libcolib.so: $(COLIB_OBJS)
49  $(BUILDSHARELIB)
50
51  example_echosvr:example_echosvr.o
52  gcc -o example_echosvr example_echosvr.o  -g -L./lib -lcolib
        -lpthread
53  example_echocli:example_echocli.o
54  gcc -o example_echocli example_echocli.o  -g -L./lib -lcolib
        -lpthread
55
```

Figure 9: An Example FRC from *libco*

File Race Condition(FRC). Figure 9 shows an FRC example in *libco*, including the report and the corresponding Makefile. For this case, a file race can occur for building target "all": the sub-targets *libcolib.a* and *example_closure* could have data race on file "libcolib.a". We expand the macros in build commands (Line 52 and 54) for better illustration. When building *example_closure*, the compiler *gcc* will access and read *libcolib.a* (option -lcolib instructs the compiler to link *libcolib.a* with *example_echosvr.o*). *libcolib.a* is created when building *libcolib.a* and there is clearly no dependency specified for these two targets. We test this FRC by building *libco* with different settings: the build fails for 4/100 times with four concurrent jobs and fails 100/100 times with more than five concurrent jobs.

Answer to RQ3 We are able to apply the VERIBUILD approach to two practical dependency-related build errors that existing approaches do not support, with less than two hundred lines of code, which, to some extent, demonstrates the good extensibility of VERIBUILD.

6 RELATED WORK

Bug detection in build scripts. Gunter [12] used a Petri net to check the correctness and detect possible optimization opportunities in Makefiles. Tamrawi et al. [37] proposed SYMAKE which uses symbolic dependency graphs to automatically detect bad smells, such as cyclic dependencies and duplicate prerequisites, in the build scripts. Xia et al. [43] and Zhou et al. [44] adopted data mining techniques to predict missing dependencies. All the above approaches only analyze the information statically available in the build scripts. Therefore, they are unable to detect many dependency issues due to the lack of the actual dependency information.

Another line of research studies is to use dynamic techniques to verify dependencies. Licker and Rice [22] proposed a fuzzing technique to detect dependency issues. It constructs a file-level dependency graph by tracing system calls of a clean build and tests the correctness of build scripts by triggering an incremental build for each file. As discussed in Section 2, build fuzzing is impractical for medium and large projects due to its overwhelming number of incremental builds. Several modern build systems such as FABRICATE [38] and MEMOIZE [29] can automatically avoid missing dependencies by monitoring the building process at runtime. Such a monitoring-based approach inevitably slows down the building

process. BAZEL [18] is a build system that creates an isolated environment for each build task. so that it can avoid writing unexpected results to the file system. ELECTRICACCELERATOR [7] automatically corrects the order of build steps at runtime, rather than detecting RDs and MDs for developers to diagnose. TUP [35], IBM CLEARCASE [16] and VESTA [41] verify dependency issues at runtime. These techniques require that the dependency issues should be triggered first, otherwise, they are unable to detect them. Different from that, our approach can detect dependency issues even before their occurrences.

Using actual dependency information together with the static dependency information in the build scripts has been proven to be effective. Bezemer et al. [6] proposed a tool to detect so-called unspecified dependencies in Make-related build systems. However, unspecified dependencies are not necessarily missing dependencies. Their approach reports 1.2 million unspecified dependencies in four projects, most of which are merely implicit dependencies intentionally omitted by the developers. Our approach also analyzes both the static and the dynamic dependencies but is different from theirs in three aspects: first, our approach performs a more fine-grained analysis that analyzes build targets rather than files; second, the UDG used in our approach encodes implicitly declared dependencies, which are missing from their approach; third, the missing and the redundant dependencies detected by our approach could be directly derived from the unspecified dependencies found by their approach.

Build error localization. There are also many research studies proposed for localizing build errors [4, 5, 24, 33, 40] and automatically fixing them [14, 23]. We believe that their techniques are complementary to ours and have the potential for greater effectiveness if combined with VERIBUILD.

Build maintenance. A massive research effort has been devoted to improving the maintainability of build systems [2, 11, 13, 37, 39]. FORMIGA [13] supports simple renaming and removal of the build targets. SYMAKE [37] focuses on the renaming and extraction of the build targets. MAKAO [2] uses an aspect-oriented approach to support adding new commands and dependencies. Vakilian et al. [39] developed tools for decomposing build specifications. CLOUDMAKE [11] from Microsoft leverages system call tracing for migrating build systems.

7 CONCLUSION

In this work, we have discussed the limitations of existing approaches to incorrect build rule detection. With the insight that existing approaches do not make good use of available static and dynamic dependency information, we have presented a unified approach, which has been demonstrated to be faster, more precise and have better bug-detection capability than the state-of-the-art. We believe that VERIBUILD is promising in industry settings.

ACKNOWLEDGMENTS

Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang

REFERENCES

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2008. The evolution of the Linux build system. *Electronic Communications of the EASST* (2008). https://doi.org/10.14279/tuj.eceasst.8.115.119

[2] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *IEEE International Conference on Software Maintenance (ICSM)*. https://doi.org/10.1109/ICSM.2007.4362624

[3] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. 2007. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. https://doi.org/10.1145/1229428.1229471

[4] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for make-based build crashes. In *Proceedings - 30th International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME.2014.87

[5] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for build code errors in makefiles. In *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*. https://doi.org/10.1145/2591062.2591135

[6] Cor Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering* (2017). https://doi.org/10.1007/s10664-017-9510-8

[7] Electric Cloud. 2020. ElectricAccelerator EMake: Speeds Up Builds and Tests. https://electric-cloud.com/plugins/directory/p/emake/ [Online; accessed 25-Aug-2019].

[8] Automake Contributors. 2012. GNU Autotools. https://www.gnu.org/software/automake [Online; accessed 25-Aug-2019].

[9] Drake. 2020. Build Cop. https://drake.mit.edu/buildcop.html [Online; accessed 25-May-2020].

[10] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* (1979). https://doi.org/10.1002/spe.4380090402

[11] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014. Automated migration of build scripts using dynamic analysis and search-based refactoring. *ACM SIGPLAN Notices* (2014). https://doi.org/10.1145/2714064.2660239

[12] Carl A. Gunter. 1996. Abstracting dependencies between software configuration items. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/250707.239129

[13] Ryan Hardt and Ethan V. Munson. 2013. Ant build maintenance with formiga. In *2013 1st International Workshop on Release Engineering, RELENG 2013 - Proceedings*. https://doi.org/10.1109/RELENG.2013.6607690

[14] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE/ACM. https://doi.org/10.1145/3180155.3180181

[15] Lorin Hochstein and Yang Jiao. 2011. The cost of the build tax in scientific software. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

[16] International Business Machines Corporation (IBM). 2020. IBM Rational Clearcase. https://www.ibm.com/us-en/marketplace/rational-clearcase [Online; accessed 25-Aug-2019].

[17] Google Inc. 2019. Actual and declared dependencies. https://docs.bazel.build/versions/master/build-ref.html#actual_and_declared_dependencies [Online; accessed 25-Aug-2019].

[18] Google Inc. 2019. Bazel - a fast, scalable, multi-language and extensible build system. https://bazel.build/ [Online; accessed 25-May-2020].

[19] Google Inc. 2019. Bazel Query. https://docs.bazel.build/versions/master/query-how-to.htmle [Online; accessed 25-Aug-2019].

[20] Lim James. 2019. Combating Flaky Builds. https://medium.com/@jimjh/combating-flaky-builds-f8aaa9ccd29a [Online; accessed 25-Aug-2019].

[21] Petr Machata Juan Cespedes. 2019. ltrace. http://man7.org/linux/man-pages/man1/ltrace.1.html [Online; accessed 25-May-2020].

[22] Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 1234–1244. https://doi.org/10.1109/ICSE.2019.00125

[23] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: How far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3293882.3330578

[24] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. https://doi.org/10.1109/SANER.2018.8330201

[25] GNU Make Manual. 2019. Auto-Dependency Generation. ftp://ftp.gnu.org/old-gnu/Manuals/make-3.77/html_node/make_43.html [Online; accessed 25-Aug-2019].

[26] GNU Make Manual. 2019. Prerequisite Types. https://www.gnu.org/software/make/manual/html_node/Prerequisite-Types.html [Online; accessed 25-Aug-2019].

[27] Ken Martin and Bill Hoffman. 2010. *Mastering CMake: a cross-platform build system*. Kitware.

[28] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. *Event London* (1989). https://doi.org/10.1.1.47.7435

[29] Bill McCloskey. 2019. memoize. https://github.com/kgaughan/memoize.py [Online; accessed 25-May-2020].

[30] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. 2014. Mining co-change information to understand when build changes are necessary. In *Proceedings - 30th International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME.2014.46

[31] Microsoft. 2020. NMAKE Reference. https://docs.microsoft.com/en-us/cpp/build/reference/nmake-reference?view=vs-2019 [Online; accessed 25-May-2020].

[32] Eric S Raymond. 2003. *The art of Unix programming*. Addison-Wesley Professional.

[33] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated Localization for Unreproducible Builds. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA, 71–81. https://doi.org/10.1145/3180155.3180224

[34] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: A case study (at google). In *Proceedings - International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2568225.2568255

[35] Mike Shal. 2009. Build system rules and algorithms. *Published online (2009)*. Retrieved July 18 (2009), 2013. http://gittup.org/tup/build_system_rules_and_algorithms.pdf

[36] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. Build Code Analysis with Symbolic Evaluation. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, 650–660.

[37] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. 2012. SYMake: a build code analysis and refactoring tool for makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 366–369.

[38] Brush Technology. 2019. fabricate. https://github.com/brushtechnology/fabricate [Online; accessed 25-May-2020].

[39] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. 2015. Automated Decomposition of Build Targets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE)* (Florence, Italy). IEEE Press, 123–133.

[40] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-break My Build: Assisting Developers with Build Repair Hints. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (2018), 41–4110.

[41] VestaSys. 2020. Vesta Configuration Management System. http://www.vestasys.org/ [Online; accessed 25-May-2020].

[42] Wikipedia contributors. 2020. List of NP-complete problems. https://en.wikipedia.org/w/index.php?title=List_of_NP-complete_problems&oldid=957698266 [Online; accessed 25-May-2020].

[43] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2014. Build system analysis with link prediction. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. https://doi.org/10.1145/2554850.2555134

[44] Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. 2014. Build predictor: More accurate missed dependency prediction in build configuration files. In *Proceedings - International Computer Software and Applications Conference (COMPSAC)*. https://doi.org/10.1109/COMPSAC.2014.12