2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Root Cause Localization for Unreproducible Builds via Causality Analysis over System Call Tracing

Zhilei Ren
Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province;
School of Software, Dalian University of Technology
zren@dlut.edu.cn

Changlin Liu
Department of Computer and Data Sciences,
Case Western Reserve University
cxl1029@case.edu

Xusheng Xiao
Department of Computer and Data Sciences,
Case Western Reserve University
xusheng.xiao@case.edu

He Jiang
School of Software,
Dalian University of Technology
jianghe@dlut.edu.cn

Tao Xie
Department of Computer Science and Technology, Peking University
taoxie@pku.edu.cn

***Abstract*—Localization of the root causes for unreproducible builds during software maintenance is an important yet challenging task, primarily due to limited runtime traces from build processes and high diversity of build environments. To address these challenges, in this paper, we propose REPTRACE, a framework that leverages the uniform interfaces of system call tracing for monitoring executed build commands in diverse build environments and identifies the root causes for unreproducible builds by analyzing the system call traces of the executed build commands. Specifically, from the collected system call traces, REPTRACE performs *causality analysis* to build a dependency graph starting from an inconsistent build artifact (across two builds) via two types of dependencies: read/write dependencies among processes and parent/child process dependencies, and searches the graph to find the processes that result in the inconsistencies. To address the challenges of massive noisy dependencies and uncertain parent/child dependencies, REPTRACE includes two novel techniques: (1) using differential analysis on multiple builds to reduce the search space of read/write dependencies, and (2) computing similarity of the runtime values to filter out noisy parent/child process dependencies. The evaluation results of REPTRACE over a set of real-world software packages show that REPTRACE effectively finds not only the root cause commands responsible for the unreproducible builds, but also the files to patch for addressing the unreproducible issues. Among its Top-10 identified commands and files, REPTRACE achieves high accuracy rate of 90.00% and 90.56% in identifying the root causes, respectively.**

***Index Terms*—Unreproducible builds, localization, system call tracing**

I. INTRODUCTION

A software build is reproducible if given the same source code, build environment, and build instructions, any user can generate bit-by-bit identical copies of all specified artifacts [1]. In this definition, the source code refers to a copy of the code checked out from the source code repository, and the build artifacts include executables, distribution packages, and file system images. Note that relevant attributes of the build environment (including build dependencies, build configuration, and environment variables) are kept as part of the input for building the artifacts. A reproducible software build plays a critical role in various important applications, such as build-environment safety, software debugging, and continuous delivery [2], [3].

TABLE I
SNIPPET OF VARIATIONS, ACCORDING TO THE ***reprotest*** TOOL CHAIN

Variation	First build	Second build
env TZ	"GMT+12"	"GMT-14"
env LANG	"C.UTF-8"	one of "fr_CH.UTF-8", "zh_CN", "es_ES", "ru_RU.CP1251", "kk_KZ.RK1048"
umask	0022	0002
filesystem	default file system	disorderfs
…	…	…

Reproducible-build validation has emerged in recent years as one important software development practice, which aims to construct an independently-verifiable bridge between the source code and the build artifacts. Many open-source software projects have initiated their validation processes, such as Debian [4], Guix [5], and F-Droid [6]. In particular, to validate the reproducibility of software packages in different build environments, variations aside from the specified build environment could be introduced deliberately. For example, ***disorderfs***, a userspace file system that introduces non-determinism into metadata[1], is used to validate whether the issue of file ordering affects the reproducibility of the build. Table I illustrates example variations introduced by the validation tool chain named ***reprotest***[2] of the Debian distribution.

Once a build is identified as unreproducible (i.e., there exists any artifact with different checksum values over build environments with variations), it is critical yet challenging to perform *causality analysis* that identifies the root causes (usually one or more build commands) for the unreproducible builds, since build processes usually produce insufficient runtime traces for locating root causes. As shown in a previous study [7], the main source of runtime traces available for

[1]https://tracker.debian.org/pkg/disorderfs
[2]https://tracker.debian.org/pkg/reprotest

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00056

locating problematic files that result in unreproducible builds is the build log, being the verbose output of the ***make*** build system. However, the build log contains only high-level build commands and cannot capture the low-level build commands invoked by the high-level build commands; these low-level build commands can play a critical role in causality analysis. For instance, consider a POSIX Shell script invoked in a Makefile. From the build log, the execution of the script could be reflected, but we cannot know what underlying build commands have been invoked inside the script. Also, the build log often contains a lot of noises for causality analysis, such as greeting information, progress indicator, and test case output. Such irrelevant information makes it difficult to extract useful information.

Another major challenge for causality analysis is to deal with the high diversity of build environments. Indeed, it is possible to instrument the build commands for tracing the dependencies between the inconsistent artifacts and the build commands for some specific build systems. However, such an intrusive approach is not practical in many industrial software projects, because modern software projects such as Linux distributions often use different types of build systems (such as ***Automake***[3] and ***CMake***[4]) for different components. Additionally, these projects use many build-maintaining scripts written in POSIX Shell, Python, Perl, etc [8], [9]. It is difficult to instrument all these scripts for tracing the executed commands.

To address these challenges, in this paper, we propose a framework, REPTRACE, that collects the *system call traces* of the executed build commands (i.e., the processes spawn from the commands) and performs causality analysis over the traces for identifying the root causes for unreproducible builds. Our work is inspired by the recent successes of system call tracing in monitoring executed commands for various research fields, such as intrusion detection [10], computational reproducibility [11], and system profiling [12]. In particular, system call tracing provides two unique benefits. First, system call tracing provides a uniform interface for monitoring the operating system, such as process control, file management, and communications. Hence, it is possible to capture more accurate information of the build process. Second, since system call tracing does not rely on certain types of build systems, it can be used in different build environments.

To conduct causality analysis with system call tracing, REPTRACE builds a dependency graph of inconsistent artifacts based on two types of dependencies, and searches the graph to identify the process that causes the inconsistencies. Specifically, REPTRACE defines two types of dependencies: (1) **read/write dependency**: two processes p_1 and p_2 are said to have the read/write dependency if p_1 writes to a file and then p_2 reads from the file; (2) **parent/child process dependency**: two processes p_1 and p_2 are said to have the parent/child dependency if p_1 spawns p_2. Based on these dependencies, REPTRACE starts from the inconsistent artifact, and then identifies the responsible processes based on the dependencies of the artifact. If there are other processes that have dependencies on the identified process, and these processes also produce inconsistent artifacts or generate inconsistent runtime values, REPTRACE continues to trace back from these processes. This tracing process stops if there are no more dependencies, and then reports the last found processes as the root causes of the unreproducible build.

However, in order to effectively identify root causes, causality analysis over system call traces needs to address two major technical challenges:

- **Massive noisy dependencies**. The raw data of system calls are noisy and of a huge volume [13]. Even only considering the read/write dependencies, the number of system call traces per package could be up to tens of thousands in our work. However, most of the system call traces are irrelevant to the root causes for unreproducible builds. For example, on average only 8.59% of the *write* system calls generate different data between different rounds of build.
- **Uncertain parent/child dependencies**. When a command spawns a new process (as commonly seen in the build process), the dependencies may or may not be carried over to the child process. Simply discarding all parent/child dependencies might fail to capture the propagation trajectories toward the inconsistent artifacts. In contrast, establishing dependencies for all parent/child processes can produce many false warnings.

To tackle these two technical challenges, REPTRACE includes two novel techniques to capture the essential dependencies relevant to unreproducible builds. To address the first challenge, REPTRACE narrows down the search scope by applying differential analysis over system call traces of multiple builds. More specifically, REPTRACE identifies the *write* system calls that output different data between different builds, and traces how these differences propagate to the inconsistent artifacts. To address the second challenge, REPTRACE computes the similarity of the runtime values (extracted from the system call arguments, see Section II) passed between parent and child processes to determine whether a dependency should be established.

To assess the effectiveness of REPTRACE, we conduct an evaluation over 180 real-world packages from the Debian repository. For the task of build-command localization, REPTRACE accomplishes accuracy rate of 66.11% for the topmost-ranked build command. If we further consider the Top-10 ranked build commands, the accuracy rate reaches 90.00%. Furthermore, REPTRACE is effective in locating the files to patch for unreproducible builds; compared with a related state-of-the-art approach [7], REPTRACE achieves 10.56% percentage improvement, considering the accuracy rate for the topmost retrieved results.

This paper makes the following main contributions:

- The REPTRACE framework, being the first to conduct system call tracing and causality analysis on the collected

[3]https://www.gnu.org/software/automake/
[4]https://cmake.org/

traces for locating root causes of unreproducible builds.
- The definition of two types of dependencies (read/write dependencies among processes and parent/child process dependencies) for causality analysis.
- Two novel techniques (differential analysis on multiple builds and similarity computation of runtime values) to address massive noisy dependencies and uncertain parent/child dependencies.
- Comprehensive evaluation on 180 real-world packages to demonstrate high effectiveness of REPTRACE and its superiority over a related state-of-the-art approach [7].

II. BACKGROUND AND MOTIVATING EXAMPLE

In this section, we first describe the background of system call tracing and our representation of the captured system calls. Then, we provide a motivating example to illustrate the causality analysis based on system call tracing. The example uses a real-world package, i.e., ***airstrike*** (0.99+1.0pre6a-7), a game packaged by the Debian repository.

A. Background and Definitions

To identify the root cause of unreproducibility, we first apply system call tracing on both rounds of reproducibility validation and collect the traces. A typical system call trace snippet of the first round of build is presented in Fig. 1. From each line of system call snippet, we can obtain the process identifier (*PID*), the parent process identifier (*PPID*), the system call name, and the arguments. We can also obtain each system call's start and end time, which is not illustrated in the figure. In this work, we are interested in the file manipulation (such as *read*, *write*, and *rename*) and the process-control-related system calls (such as *execve*). With these system calls, we could gain more insights into the build process. For instance, Table II shows the processes that have the last access time to the inconsistent artifacts, as well as typical build commands. Also, with the *PID* and *PPID* information, we are able to restore the process tree structure of the build.

To model the reproducibility validation, two rounds of build are introduced as B_1 and B_2, respectively. Each round of build comprises a sequence of processes, e.g., $\{P_1, P_2, P_3, ...\}$. Specifically, a process is represented as a tuple $\langle \mathit{PID}, \mathit{PPID}, \mathit{slist}\rangle$, where the first two fields are self-explanatory, and the *slist* field indicates a list of system call traces. Each system call $s \in \mathit{slist}$ is represented by a tuple $\langle \mathit{type}, \mathit{start\text{-}time}, \mathit{end\text{-}time}, \mathit{source}, \mathit{target}, \mathit{data}\rangle$. For the *type* field, we are interested in a subset of system calls related to file manipulation and process control, including *read*, *write*, *rename*, *execve*, *open*, and *fcntl*. The *start-time* and the *end-time* fields represent the starting and end time of the system call. The remaining fields are system call specific:

1) *read* represents the *read* system call and its variants, such as *readv* and *preadv*. The *source* field specifies the file to read, and the *data* indicates the bytes read from *source*.
2) *write* represents the *write* system call and its variants, such as *writev* and *pwritev*. The *target* field specifies the file to write, and the *data* indicates the bytes written to *target*.

```
4213 4212 execve("/usr/bin/ld", ["/usr/bin/ld", [...] "-o", "airstrike" ...
4212 4211 execve("/usr/bin/cc", ["cc", "-o", "airstrike", "-g", "-O2" ...
4000 3999 execve("/usr/bin/make", ["make", "-C", "src" ...
4000 3999 write(1<[...].log>, "cc -o airstrike [...] ./players.o ./airstrike.o ...
4028 4000 execve("/bin/sh", ["/bin/sh", "-c", "cc -o airstrike ...
4000 3999 read(8<pipe:[31387067]>, "./players.c\n./airstrike.c\n ...
4002 4000 write(1<pipe:[31387067]>, "./players.c\n./airstrike.c\n ...
```

Fig. 1. System call trace snippet for ***airstrike***

3) *rename* represents the system calls of *rename*, *renameat*, *renameat2*, and *linkat*. The *source* and *target* fields are used to specify the file names for renaming (changing from *source* to *target*).
4) *execve* represents the family exec system calls, i.e., *execve* and *execveat*. The *data* field represents the build command invoked, including both the executable and the arguments.
5) *open* represents the system calls of *open*, *openat*, and *creat*. The *source* and *data* fields indicate the file and the corresponding flags assigned to the file.
6) *fcntl* manipulates a file descriptor. The *source* and the *data* fields indicate the file and the corresponding flags assigned to the file.

Note that we use an underscore (_) to denote that a specific field of a system call is ignored. For example, for the *rename* system call, only the *source* and *target* fields are used, and the *data* field is ignored.

Definition 1 (runtime value): Given a process P, its runtime value is defined as a set $V = \{s.\mathit{data} | s \in P.\mathit{slist}, s.\mathit{type} = \mathit{read}, \mathit{write}, \text{or } \mathit{execve}\}$. The underlying motivation of runtime value is that the *data* of the *read*, *write*, and *execve* system calls play an important role during the propagation of the inconsistencies.

Definition 2 (read/write dependency): Given two processes $\langle \mathit{PID}_1, \mathit{PPID}_1, \mathit{slist}_1\rangle$, $\langle \mathit{PID}_2, \mathit{PPID}_2, \mathit{slist}_2\rangle$ of the same build, a read/write dependency ($\mathit{PID}_1 \xrightarrow{f} \mathit{PID}_2$) is established if (1) $\exists\langle \mathit{read}, st_1, et_1, f, \_, \mathit{data}_1\rangle \in \mathit{slist}_1$, $\langle \mathit{write}, st_2, et_2, \_, f, \mathit{data}_2\rangle \in \mathit{slist}_2$, such that $et_1 > et_2$, or if (2) $\exists\langle \mathit{read}, st_1, et_1, f_1, \_, \mathit{data}_1\rangle \in \mathit{slist}_1$, $\langle \mathit{write}, st_2, et_2, \_, f, \mathit{data}_2\rangle \in \mathit{slist}_2$, and another system call $\langle \mathit{rename}, st_3, et_3, f, f_1, \_\rangle$ from any process of the same build, such that $et_1 > et_3 > et_2$ [5].

Definition 3 (parent/child process dependency): Given two processes P_1 and P_2 with *PID*s p_1 and p_2, respectively, of the same round of build, if $P_1.\mathit{PPID} = P_2.\mathit{PID}$, there exists a parent/child process dependency between the two processes, denoted as $p_1 \Rightarrow p_2$.

B. Motivating Example

Based on these notations and definitions, we next present a running example to motivate REPTRACE. With the captured

[5]In this definition, only single rename is considered in this type of dependency. It is straightforward to extend to the case of multiple renames. In this work, no significant difference is observed between the two variants.

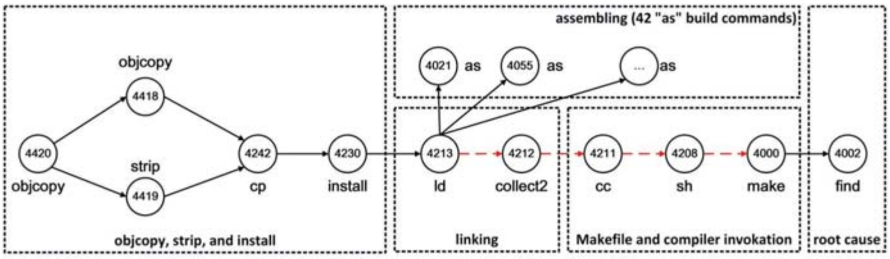


Fig. 2. Dependency graph for ***airstrike***

TABLE II
INCONSISTENT ARTIFACTS AND TYPICAL BUILD COMMANDS FOR THE DEPENDENCY GRAPH OF ***airstrike***

PID	Artifact
4420	./usr/games/airstrike

PID	Build command
4420	objcopy [. . .] debian/airstrike/usr/games/airstrike"
4419	strip [. . .] debian/airstrike/usr/games/airstrike"
4242	cp –reflink=auto -a, debian/tmp/usr/games, debian/airstrike//usr/
4240	install airstrike [. . .]/usr/games/airstrike
4213	ld [. . .] -o airstrike [. . .] ./players.o ./airstrike.o [. . .]
4021	as -I [. . .] -o players.o /tmp/cc3wloBL.s
4142	as -I [. . .] -o airstrike.o /tmp/ccLbhlnX.s
4212	collect2 [. . .] -o airstrike [. . .] ./players.o ./airstrike.o [. . .]
4208	sh -c cc -o airstrike [. . .] ./players.o ./airstrike.o [. . .]
4000	make -C src airstrike
4002	find . -name *.c

system call traces, we are able to capture the dependencies between the processes within the same build. The dependency graph for ***airstrike*** contains in total 380 nodes. Each of node represents a process. Starting from the process in which the inconsistent artifact *./usr/games/airstrike* is last accessed (process with *PID* 4420), we are interested in how the inconsistency is introduced by the root cause, and how it is propagated.

Fig. 2 shows part of the dependencies between the processes for ***airstrike***. In the figure, the solid arrows and the dashed arrows indicate the read/write dependencies and the parent/child process dependencies, respectively. By traversing the dependency graph, we can locate the root cause for the unreproducibility. However, there may exist many irrelevant dependencies in the graph. The reason is that the criterion for establishing dependencies between process is loose, and does not take the read/written *data* into consideration. We should note that not all these processes in the graph introduce inconsistencies between the two builds. For example, consider the build command ***ld*** in the process with *PID* 4213, which is the GNU linker to create an executable from object files. With the dependency rule described in **Definition 2**, we have to further investigate all the processes that write to the associated object files. There are 42 object files during the link stage, leading to 42 edges in the dependency graph. However, in this case, all the object files are actually consistent between different builds, implying that the corresponding edges all represent irrelevant dependencies.

In fact, the inconsistency for ***airstrike*** results from the order of the linker arguments; the order is propagated from its parent process (with *PID* 4212). The corresponding build command is ***collect2***, a ***GCC*** utility to arrange to call various initialization functions, and invoke the linker. By carefully inspecting the traces, we find that the dependency between this pair of processes could be revealed from the *text similarity* between their build command arguments (4213 $\Rightarrow$ 4212). Following this clue, we could traverse to the process with *PID* 4208 (***cc***). At this point, the hint for further traversal (4208 $\Rightarrow$ 4000) comes from the *data* field of the *write* system call for the ***make*** command (see Table II). Finally, we can discover a dependency toward a ***find*** command (4000 $\xrightarrow{f}$ 4002), where f=pipe:[31387067], indicating that the ***make*** command reads the output of ***find*** through a pipeline. To this end, we could gain better understanding for the root cause of the inconsistent artifact, i.e., the file traversal order of ***find*** is not guaranteed to be deterministic. Consequently, because the link order relies on the output of ***find***, the build artifact turns out to be unreproducible.

Based on these observations, REPTRACE filters out the build commands that write identical *data* between the two builds of the validation; this filtering can effectively simplify the dependency graph. Second, to identify the parent/child process dependencies, we calculate the similarity of the runtime values of the parent process and the child process, and establish dependencies only for those parent/child processes that share similar runtime values. In this way, we could identify the relevant dependencies without introducing too many irrelevant dependencies.

III. OUR REPTRACE FRAMEWORK

In this section, we describe the design and implementation of the proposed REPTRACE framework. As illustrated in Fig. 3, given the source package, we first adopt the tool chain of reproducibility validation to build the source code under the build environments with variations. During the build process, we collect the system call traces of the two builds using ***strace*** [14], a popular diagnosis utility. Then, we construct the dependency graph based on the sliced, abstracted system call traces, which are produced by applying differential analysis over the two sets of system call traces. After that, we intend to augment the dependency graph by detecting the parent/child process dependencies with runtime values. By improving the dependency graph with the runtime-value-induced dependencies, the root causes could be better located with the traversal over the improved dependency graph.

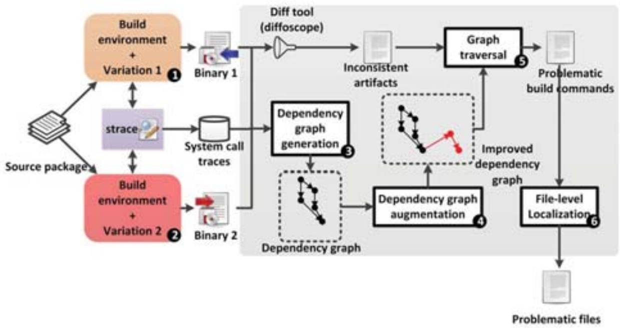


Fig. 3. The REPTRACE framework

A. Dependency Graph Generation and Augmentation

As mentioned in Section I, a major challenge of analyzing system call traces lies in the massive volume of the gathered data. To extract the useful information, meanwhile reducing the noises, our key idea is to perform differential analysis of the system call traces. In this work, because we are interested in how the inconsistencies are generated between multiple builds, and how these inconsistencies are propagated to the build artifacts, we construct the dependency graph based on the differences extracted from the system call traces. In particular, in the dependency graph produced by REPTRACE, the read/write dependencies are replaced with the difference-induced dependencies, as defined below:

Definition 4 (difference-induced dependency): For the two builds B_1 and B_2 of the reproducibility validation, *write-diff* is denoted as a set $\bigcup_{P \in B_1} \bigcup_{\substack{s \in P.slist \\ s.type = write}} \textit{md5}(s.data) - \bigcup_{P \in B_2} \bigcup_{\substack{s \in P.slist \\ s.type = write}} \textit{md5}(s.data)$. Given two processes $\langle \textit{PID}_1, \textit{PPID}_1, \textit{slist}_1 \rangle \in B_1$, $\langle \textit{PID}_2, \textit{PPID}_2, \textit{slist}_2 \rangle \in B_1$, a difference-induced dependency (DID, denoted as $\textit{PID}_1 \overset{f}{\hookrightarrow} \textit{PID}_2$) is established, if $\textit{PID}_1 \overset{f}{\to} \textit{PID}_2$, and $\exists \ \langle \textit{write}, st, et, \_, f, \textit{data} \rangle \in \textit{slist}_2$, such that $\textit{md5(data)} \in \textit{write-diff}$.

Algo. 1 shows the pseudo code of the dependency graph construction based on difference-induced dependencies. The proposed algorithm comprises two phases. First, the set *write-diff* is calculated, based on the *data* field of each *write* system call. The unique feature of our graph construction is that, to reduce the noises within the system calls, the dependency propagation focuses on the inconsistencies generated by the *write* system calls between builds. Then, each pair of *write* and *read* system calls are examined whether a difference-induced dependency should be established between the corresponding processes. In particular, for each *write* system call s_1 with respect to *write-diff* (Line 6), we examine whether there exists a *read* system call reading from $s_1.dest$ after time $s_1.et$. If so, a dependency is established (Lines 8–12). Similarly, if the file $s_1.dest$ is renamed by a *rename* system call s_3 to $s_3.dest$, and later read by a *read* system call, a dependency should also be established (Lines 13–18).

Algorithm 1: Difference-based dependency graph generation

Input: System call traces for *read*, *write*, and *rename*
1 **begin**
2     $G \leftarrow emptygraph$
3     $\textit{write-hash}_1 \leftarrow \bigcup_{P \in B_1} \bigcup_{\substack{s \in P.slist \\ s.type = write}} \textit{md5}(s.data)$
4     $\textit{write-hash}_2 \leftarrow \bigcup_{P \in B_2} \bigcup_{\substack{s \in P.slist \\ s.type = write}} \textit{md5}(s.data)$
5     $\textit{write-diff} \leftarrow \textit{write-hash}_1 - \textit{write-hash}_2$
6   **for** *write* system call s_1 where $\textit{md5}(s_1.data) \in \textit{write-diff}$ **do**
7         $\textit{pid-write} \leftarrow \textit{pid-of}(s_1)$
8   **for** *read* system call s_2 **do**
9   **if** $s_2.src = s_1.dest$ ***and*** $s_2.et > s_1.et$ **then**
10                 $\textit{pid-read} \leftarrow \textit{pid-of}(s_2)$
11                 $\textit{add-edge}(G, \textit{pid-read}, \textit{pid-write})$
12   **end**
13   **for** *rename* system calls_3 **do**
14   **if** $s_3.src = s_1.dest$ ***and*** $s_3.dest = s_2.src$ ***and*** $s_2.et > s_3.et > s_1.et$ **then**
15                     $\textit{pid-read} \leftarrow \textit{pid-of}(s_2)$
16                     $\textit{add-edge}(G, \textit{pid-read}, \textit{pid-write})$
17   **end**
18   **end**
19   **end**
20   **end**
21   **return** G
22 **end**

Furthermore, to tackle the challenge of the uncertain parent/child process dependency, REPTRACE utilizes the text similarity of the runtime values. As discussed in Section II, the runtime values passed between the processes can be used to reveal the dependencies. In particular, for script-based build systems, the runtime values are mostly in the format of plain text. Consequently, we could leverage text similarity to make decisions on whether dependencies should be established. Specifically, the relevance value is calculated as follows.

Definition 5 (relevance value): Given two processes with *PID*s p_1 and p_2, each with a sequence of runtime values $V_1 = \{v_{11}, v_{12}, \ldots v_{1m}\}$ and $V_2 = \{v_{21}, v_{22}, \ldots v_{2n}\}$, the relevance between the two processes is calculated as

$$relevance(p_1, p_2) = \max_{v_1 \in V_1, v_2 \in V_2} \{\max\{cosSim(v_1, v_2), lcsSim(v_1, v_2)\}\}, \tag{1}$$

where $cosSim$ and $lcsSim$ represent the cosine-based [15] and the longest-common-substring-based [16] similarity, respectively. Note that for $lcsSim$, we consider the longest common substring percentage, with the value ranging within $[0, 1]$. The motivation behind the similarity measurement is that the length of the runtime values might be of arbitrary length. Hence, using only one type of similarity might not be effective for various cases. Specially, we skip the pairs of runtime values when the runtime values contain binary data by assigning 0 to the similarity value. With the relevance value, the runtime-value-induced dependency is defined as follows.

Definition 6 (runtime-value-induced dependency): Given two processes with *PID*s p_1 and p_2 of the same round of build, there exists a runtime-value-induced dependency (RID, denoted as $p_1 \overset{I}{\Rightarrow} p_2$) if $p_1 \Rightarrow p_2$, and the relevance value between the processes is larger than the pre-defined threshold.

Based on **Definition 6**, the dependency graph constructed with Algo. 1 can be further improved based on Algo. 2. For

Algorithm 2: Dependency graph augmentation

```
Input: Dependency graph G, relevance threshold THRESHOLD
begin
    G' ← G
    for node ∈ G' do
        PID ← pid-of(node)
        parent-pid ← parent-of(node)
        Calculate relevance with Eq. 1.
        if relevance > THRESHOLD then
            add-edge(G', PID, parent-pid)
        end
    end
    return G'
end
```

each node that represents a process, we calculate its relevance value with its parent process. If the relevance value is greater than the given threshold, a runtime-value-induced dependency should be established.

Running example: Using the package ***airstrike***, we explain how the two mechanisms work. First, when evaluating the process with *PID* 4213 (the ***ld*** command), for those input files (of the ***ld*** command) that are consistent between builds, it is obvious that their corresponding *write* system calls are associated with the same hash values. Hence, these processes could be neglected. Second, similarly, to demonstrate that the parent/child process dependency works, consider the child process (with *PID* 4213) and the parent process (with *PID* 4212). The relevance value calculated by Eq. 1 is 0.9993, which provides strong evidence that there should be a dependency between the two processes.

B. Graph-Traversal-based Causality Analysis

After constructing and augmenting the dependency graph, REPTRACE traverses the graph, searching for the root causes for the unreproducible builds. As shown in Algo. 3, REPTRACE starts the traversal from the nodes that represent the processes directly accessing the inconsistent artifacts. From these nodes, REPTRACE performs a breadth-first search, and obtains a set of nodes without outgoing edges to other unvisited nodes in the graph (Line 2). Since the edges in the dependency graph indicate the trajectories of the inconsistency propagation, these nodes indicate that the inconsistency propagation stops at these nodes, i.e., no more inconsistencies propagated to other processes. With these nodes obtained, REPTRACE then ranks them based on their relevance values among other nodes in the dependency graph (Lines 6–11). The higher the accumulated relevance value is, the higher probability the corresponding nodes would be the root causes.

Finally, to realize the file-level localization, we start from the ranked list of build commands retrieved by Algo. 3, in search of the most relevant files. More specifically, based on the preliminary investigation, two different paradigms of patches are identified.

- **Case 1**: for those packages in which scripts are responsible for the unreproducibility, such as the wildcard function of Makefiles and the hash-table traversal of Perl scripts, the scripts are to be patched, being opened in the same process as the one where the root causes are identified.

Algorithm 3: Graph-traversal-based root-cause localization

```
   Input: Improved dependency graph G
1  begin
2      node-set ← breadth-first-search(G)
3      for node m ∈ node-set do
4          PID_m ← pid-of(m)
5          node-weight[PID_m] ← 0
6          for node n ∈ G do
7              PID_n ← pid-of(n)
8              node-weight[PID_m] ←
                 node-weight[PID_m] + relevance(PID_m, PID_n)
9          end
10     end
11     ranked-list ← sort(node-set, node-weight)
12     return ranked-list
13 end
```

Algorithm 4: File-level localization

```
Input: Node set node-set, Weights for the nodes node-weights, File set file-set
begin
    for file f ∈ file-set do file-weight[f] ← 0
    for node m ∈ node-set do
        PID_m ← pid-of(m)
        switch typeof(m) do
            case 1: search in the current process do
                for file f opened with CLOEXEC flags do
                    file-weight[f] ←
                      file-weight[f] + node-weight[PID_m]
                end
            end
            case 2: search in the parent process do
                PPID_m ← parent-of(m)
                t_1 ← get-execve-text(m)
                for file f opened with CLOEXEC in process PPID_m do
                    t_2 ← get-text(f)
                    sim ← max{cosSim(t_1, t_2), lcsSim(t_1, t_2)}
                    file-weight[f] ←
                      file-weight[f] + node-weight[PPID_m] × sim
                end
            end
        end
    end
    ranked-file-list ← sort(file-set, file-weight)
    return ranked-file-list
end
```

- **Case 2**: for the build commands that may introduce inconsistencies, such as the ***gzip*** and the ***date*** commands. In this case, the scripts to be patched are typically opened in the parent process of the identified process. For example, in the motivating example ***airstrike***, the inconsistency is introduced by the ***find*** command. However, the file to be patched is the Makefile in which ***find*** is invoked (see Fig. 4).

To distinguish the two cases, we adopt a heuristic rule based on the flags associated to each opened file. In particular, the scripts are typically opened with the CLOEXEC flags (FD_CLOEXEC or O_CLOEXEC), indicating that the files are to be closed automatically after successful execve system calls. During our preliminary experimentation, we observe that the CLOEXEC flags are generally effective in classifying the scripts and the other files, with two exceptions, i.e., the processes invoking Python scripts or the ***tar*** compressing utility, which are processed in a specialized way. With the heuristic classifying rule, the process of localization for the file to patch is described in Algo. 3.

Running example: For the package ***airstrike***, after obtaining the dependency graph, the root cause for the unreproducible build can be found by traversing the graph. As shown in Fig. 2,

```
--- a/src/Makefile
+++ b/src/Makefile
@@ -2,7 +2,7 @@
 #
 CFLAGS += $(shell sdl-config --cflags) -Isprite -I. -Isupport -DINLINE=inline
 CFLAGS += `dpkg-buildflags --get CFLAGS`
-CFILES:= $(shell find . -name '*.c')
+CFILES:= $(sort $(shell find . -name '*.c'))
 OBJECTS:= $(CFILES:.c=.o)
```

Fig. 4. Patch snippet for ***airstrike***

we can observe that the node with zero outgoing unvisited edge (process with *PID* 4002, the ***find*** command) is the root cause. Furthermore, since no file is opened with the CLOEXEC flags in the process with *PID* 4002, REPTRACE checks its parent process, and locates the file to be patched (*src/Makefile*), which is shown in Fig. 4.

IV. EVALUATION

In this section, we apply REPTRACE on real-world software packages and evaluate the effectiveness of REPTRACE. We seek to investigate the following research questions (RQs):

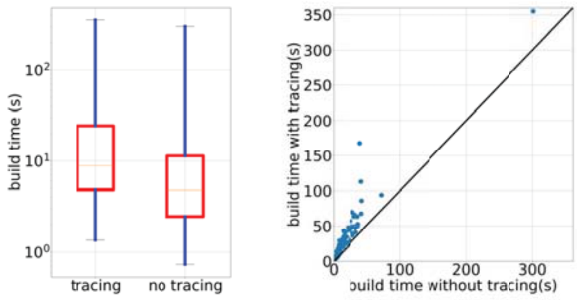
1) **RQ1**: Is REPTRACE effective in locating the root causes for unreproducible builds?
2) **RQ2**: How effectively can the DID and RID mechanisms improve the construction of dependency graphs?
3) **RQ3**: Is REPTRACE sensitive to the parameter in the runtime-value-induced dependency?
4) **RQ4**: Is REPTRACE helpful in locating the problematic files to patch?

Among these RQs, RQ1 evaluates the ability to accurately identify the root causes for unreproducible builds, because the command-level localization is the unique feature of REPTRACE. In particular, by comparing various variants of REPTRACE, we intend to examine how each component contributes to REPTRACE. RQ2 evaluates the impact of the two mechanisms on the search space. By comparing the statistics of the dependency graphs induced by the different variants of REPTRACE, we could gain more insights into both the DID and RID mechanisms. RQ3 evaluates the sensitivity of the parameter on REPTRACE. Finally, RQ4 evaluates the ability of file-level localization of REPTRACE by comparing with the best known results.

A. Evaluation Setup

REPTRACE is implemented in Java 1.8, and the evaluation is conducted on an Intel Xeon 2.5 GHz server with 16 GB memory, running Debian 9.6.

Metrics. To evaluate the effectiveness of REPTRACE, we measure the accuracy rate, precision, recall, F-1 score, and Mean Reciprocal Rank (MRR) in identifying root causes for unreproducible builds. The metrics are computed by examining the ranked build commands (RQ1) and files (RQ4) returned by REPTRACE. The Top-N build commands/files in the ranked list are called the retrieved list, and are compared with the relevance list to compute the precision, recall, and F-1 score (represented using *P@N*, *R@N*, and *F-1@N*, respectively). In particular, Top-N accuracy rate, e.g., *A@N*, is used to measure the percentage of packages for which the Top-N list provides



(a) Box-plot statistics (b) Comparison of build time

Fig. 5. Comparison of build time statistics

at least one problematic command/file [17]. Besides, MRR is an aggregate metric to evaluate the quality of the retrieved results.

Tools under comparison. In our evaluation, we compare REPTRACE with a set of variants. First, a set of three variants of REPTRACE are chosen, each considering part of the mechanisms of REPTRACE. For example, we denote REPTRACE(¬DID) as the variant of REPTRACE in which the difference-induced dependency is not employed. There are also two other variants denoted as REPTRACE(¬RID) and REPTRACE(¬DID, ¬RID), in which parent/child dependencies are not considered. With these variants, we investigate how the proposed mechanisms collaborate as an integrated framework. Second, we compare REPTRACE with REPLOC, the state-of-the-art tool for file-level localization [7].

Dataset. We use a set of 180 packages from the Debian repository, following previous work [7] as our evaluation dataset. The reasons that the scale of the dataset is relatively small are as follows. First, due to the evolution of the Debian repository, especially the build tool chain and the build dependencies, some old packages used in the previous work could not be built from source. Second, due to the necessity of manual annotation for the root causes, to evaluate the ability of causality analysis, we do not consider all the packages as in the previous work. Besides, since we focus on the identification of the root causes, which are represented as build commands, we do not consider the packages for which the patches are within source code.

In the dataset, the root causes cover the following categories: timestamp (such as ***gzip***, ***date***, and ***tar*** that capture the current date and/or time), randomness (such as dict/hash-table traversal of Python and Perl scripts), file ordering (such as ***find*** in ***findutils*** and the ***wildcard*** issue of ***make***), locale (such as ***sort*** and ***lynx*** without setting the locale environment variable), uname and hostname (***uname*** and ***hostname*** that capture the system information).

To construct the ground truth for evaluating causality analysis, we check the *execve* system call traces that match the patches obtained from the bug-tracking system of Debian. For each package, we check not only according to the build command line text, but also the context indicated by the path from the root of the process tree to the problematic build

TABLE III
RESULTS OF REPTRACE AND OTHER APPROACHES FOR THE COMMAND-LEVEL LOCALIZATION TASK

Approach	*A@1*	*A@5*	*A@10*	*P@1*	*P@5*	*P@10*	*R@1*	*R@5*	*R@10*	*F-1@1*	*F-1@5*	*F-1@10*	*MRR*
REPTRACE(¬DID, ¬RID)	0.0944	0.1056	0.1500	0.0944	0.0300	0.0200	0.0824	0.1014	0.1444	0.0850	0.0431	0.0336	0.1042
REPTRACE(¬DID)	0.0389	0.0833	0.1333	0.0389	0.0278	0.0200	0.0280	0.0787	0.1259	0.0306	0.0366	0.0322	0.0663
REPTRACE(¬RID)	0.6056	0.6556	0.6556	0.6056	0.2533	0.1461	0.3916	0.5841	0.5912	0.4401	0.3040	0.1921	0.6306
REPTRACE	0.6611	0.8944	0.9000	0.6611	0.3156	0.1839	0.4524	0.8129	0.8319	0.4993	0.3960	0.2510	0.7672

TABLE IV
RESULTS OF WILCOXON SIGNED RANK TEST FOR THE COMMAND-LEVEL LOCALIZATION TASK

Metrics	REPTRACE *vs.*	*p*-value	Effect size
P@1	REPTRACE(¬DID,¬RID)	<0.0001	0.9808
	REPTRACE(¬DID)	<0.0001	0.9333
	REPTRACE(¬RID)	0.1228	0.2381
R@1	REPTRACE(¬DID,¬RID)	<0.0001	0.9707
	REPTRACE(¬DID)	<0.0001	0.9236
	REPTRACE(¬RID)	0.0372	0.3555

command. Meanwhile, for the file-level localization, we adopt an approach in the literature [7], i.e., we extract the file names from the patches as the ground truth.

Overhead of system call tracing. As REPTRACE is built upon system call tracing, we measure to what extent system call tracing slows down the build process. We compare the statistics of the build time under two circumstances, i.e., with or without system call tracing. In Fig. 5, the box-plots and the scatter plot represent the distribution of build time over all the packages. From the figure, we observe that when using system call tracing, build time increases accordingly. For the two cases, the median build time is 4.77s and 8.84s, respectively. Meanwhile, the maximum build time for the two circumstances is of the same order of magnitude. Such results indicate that the overhead of system call tracing is acceptable for industrial-level builds.

B. RQ1: Command-Level Localization

As discussed in Section III, a unique feature of REPTRACE lies in its ability to locate the root causes for unreproducible builds. Prior to this work, the localization task realized by REPLOC [7] is mainly at the file level. Hence, the guidance toward the patch of the unreproducible builds tend to be limited. In contrast, with the system call tracing, especially the *data* provided by the *execve* system call, REPTRACE is able to identify the potential build commands that are responsible for the unreproducible issues.

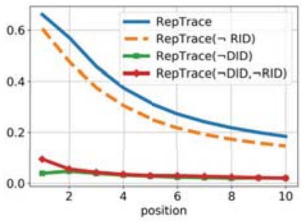
In this RQ, we compare the results of causality analysis in Table III. The table is organized as follows. The first column represents the names of the approaches in comparison, including REPTRACE and its three variants. Then, Columns 2–14 indicate the measurements employed to evaluate each approach, i.e., the accuracy rate, precision, recall, F-1 score, and MRR.

From the table, we could observe that REPTRACE is able to effectively locate the root causes responsible for the unreproducibility. Especially, when considering the topmost retrieved build command, REPTRACE is able to achieve an accuracy rate of 0.6611. The accuracy rate increases to 0.9000 if we consider the Top-10 results, implying that for 90.00% of the packages, we can obtain at least one build command that is among the root causes by traversing the Top-10 results. In contrast, the results for the variants of REPTRACE are not so promising. We should note that, from the table, we observe that the precision and F-1 score values are not very high. The reason for the low values of precision and F-1 score might be that, for the unreproducible packages, the number of processes that construct the root causes is relatively small. Within the dataset, there are 99 packages for which there is single build command that causes unreproducibility, and the average number of root causes is 3.41. Consequently, the precision value for the Top-10 results tends to be low, also influencing the F-1 score. Under such circumstance, the MRR metric reflects the ability to rank root causes to the top of results. From the table, we observe that REPTRACE is able to achieve the best MRR.

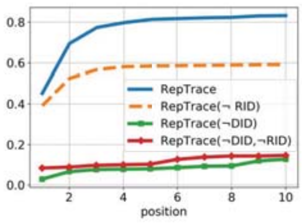
To gain higher confident on drawing conclusion from the comparison results, we employ the nonparametric Wilcoxon signed rank test. For the null hypothesis, we assume that there exists no significant difference with respect to the results obtained by the approaches under comparison. Table IV shows the comparison results, organized as follows. The first column indicates the metrics over which the comparison is conducted. The second column specifies the approaches against which REPTRACE is compared. The third and fourth columns present the *p*-value and effect size (also known as the rank-biserial correlation) [18], respectively. From the comparison results, we observe that under each comparison scenario except when comparing REPTRACE with REPTRACE(¬RID) over the *P@1* metric, the null hypothesis is rejected, with p-value < 0.05. This observation confirms that the DID mechanism contributes more to the performance.

Fig. 6 shows the results obtained by REPTRACE and its variants, against the length of the retrieved list. We consider the precision and recall as the measurements. When we compare the behavior of the variants, we could measure the improvement brought by each mechanism. For example, when we compare REPTRACE(¬DID, ¬RID) with REPTRACE(¬RID), we could see that for both the measurements, the curves for REPTRACE(¬DID, ¬RID) are always below those of REPTRACE(¬RID). This observation confirms the contribution of the DID mechanism, which not only provides a smaller dependency graph (see RQ2 for more discussion), but also helps locate the root cause more accurately. A similar observation could be made when we compare REPTRACE(¬DID) and REPTRACE. Furthermore, to understand whether the RID mechanism works, we compare REPTRACE with REPTRACE(¬RID). From Fig. 6, we could see that REPTRACE outperforms REPTRACE(¬RID).

Answer to RQ1: REPTRACE is able to effectively identify



(a) Trend for precision



(b) Trend for recall

Fig. 6. Comparison of REPTRACE and its variants

the root causes that are responsible for unreproducible builds, and these root causes are helpful in understanding why reproducibility validation fails.

C. RQ2: Impacts on Dependency Graph Construction

In REPTRACE, there are two main mechanisms, i.e., the reduction based on differential analysis, which intends to shrink the scale of search space, and the runtime-value-based dependency identification, which may enlarge the dependency graph. Hence, in this RQ, we analyze the statistics of the dependency graphs constructed by REPTRACE and its variants, to explore whether the DID mechanism is able to achieve the reduction of search space and yet preserve the precision in causality analysis.

To gain an intuitive understanding of the influence of the two proposed mechanisms, in Fig. 7, we present the comparison of the graph statistics of the dependency graph generated by the variants of REPTRACE, respectively. For each variant, we report the statistics of the dependency graph for all the packages, to reflect the influence of each mechanism. For each sub-figure, we plot the distribution of typical properties in log scale, including the number of nodes (*num_nodes*), number of edges (*num_edges*), average node degree (*avg_degree*), and maximum node degree (*max_degree*). All the statistics are illustrated as box-plot.

From the figure, we could observe the following two interesting phenomena. On one hand, when comparing Fig. 7(a) and Fig. 7(b), we could see the reduction effect of the DID mechanism. Without the DID-based reduction mechanism, there are on average 169.37 nodes in the dependency graph. Meanwhile, with the reduction, there are on average 15.64 nodes in the dependency graph, being of a much smaller scale. Also, for other graph attributes, similar phenomena could be observed. For instance, the maximum node degree of the graph for REPTRACE(¬DID, ¬RID) is larger than that for REPTRACE(¬RID), implying that without the DID mechanism, there may exist nodes with more dependencies. Consequently, the possibility of incorporating irrelevant dependencies may also increase. This observation to some extent explains why the results of the variants without DID are not satisfying in RQ1.

On the other hand, when comparing Fig. 7(b) and Fig. 7(d), we could observe that if the RID mechanism is considered over the DID-reduced dependency graph, there are not many nodes and edges introduced by the runtime-value-induced dependency mechanism. Hence, the overhead caused by the runtime-value-induced dependencies is in general acceptable. In contrast, when comparing Fig. 7(a) and Fig. 7(c), we observe a drastic increase in attribute values of dependency graph, implying that if the RID mechanism is considered over the dependency graph without reduction, the corresponding graph would be much more complex. This observation to some extent explains why REPTRACE(¬DID) performs the worst among the variants.

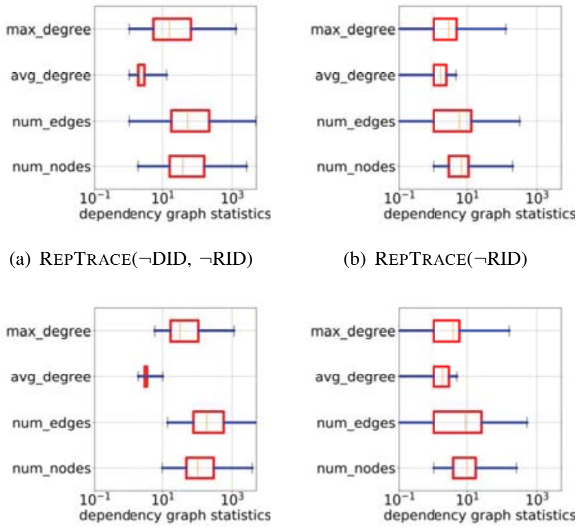
Furthermore, Fig. 8 shows the distribution of the execution time in log scale for REPTRACE and its variants. From the figure, we could observe that REPTRACE(¬RID) is the fastest variant, with median execution time of 5.88s. The reason is that the scale of the dependency graphs for this variant is smaller than other variants. REPTRACE is slower, with median execution time of 9.51s, but is within the same order of magnitude. In contrast, the two variants without DID are much slower. In particular, REPTRACE(¬DID) is the least efficient variant in comparison, due to the lack of reduction realized by the DID mechanism, and the extra dependencies introduced by the RID mechanism. This observation also conforms with Fig. 7(c), which presents the statistics of the most complex dependency graph.

Answer to RQ2: In this RQ, we confirm that the DID mechanism is able to effectively reduce the search scope of the localization task. Also, the extra edges introduced by the RID mechanism is acceptable when the DID mechanism is applied. With the dependencies induced by the differences of the *write* system call and the runtime values, REPTRACE is able to achieve median execution time of 9.51s.

D. RQ3: Parameter Sensitivity Analysis

As mentioned in Section III-A, we introduce a threshold in the RID mechanism, to detect the potential dependencies between parent processes and child processes. Hence, we shall evaluate REPTRACE's sensitivity to the threshold. Fig. 9 shows the results of the sensitivity analysis over a subset of the 40 randomly selected packages. The figure is organized as follows. The x-axis represents the value of the parameter, which ranges from 0 to 1, with the step of 0.10. The y-axis indicates the quality measurement, i.e., the precision and recall considering the Top-1 result.

From the figure, we could observe that, REPTRACE is not very sensitive to the threshold, in terms of both measurements. For example, for all the parameter values, the precision value



(a) REPTRACE(¬DID, ¬RID) (b) REPTRACE(¬RID)

(c) REPTRACE(¬DID) (d) REPTRACE

Fig. 7. Dependency graph statistics for variants of REPTRACE

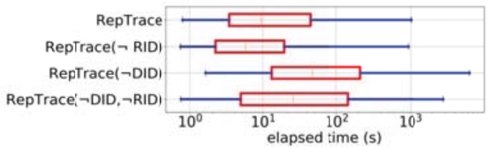


Fig. 8. Comparison of execution time for variants of REPTRACE

is always above 70.00%. Also, both the recall and accuracy rate reach the best results around $[0.30, 0.70]$.

Answer to RQ3: REPTRACE is not very sensitive to the parameter, and generalizes well over different packages. Hence, for the other parts of the evaluation, the parameter value is assigned with 0.50.

E. RQ4: File-Level Localization

In this RQ, we evaluate whether REPTRACE is effective in locating the problematic file, in which the unreproducible issues should be patched. Specifically, we report the results obtained by REPTRACE and other approaches under comparison in Table V. The table is organized similarly as Table III, except that REPLOC is also considered.

From the table, we can observe that REPTRACE is able to rank the relevant files at the top of the retrieved list. Compared with REPLOC, the Top-1 accuracy rate is 0.6667, which is much higher than the results achieved by REPLOC. The underlying reason might be that with the system-call-based dependency graph, REPTRACE is able to accurately locate the build commands for which there exists at least one path in the dependency graph leading to the inconsistent artifacts. Consequently, the file ranking based on these build commands could provide valuable hints toward the problematic files to be patched. In contrast, REPLOC relies on the

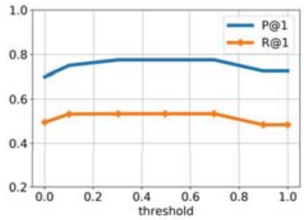


Fig. 9. Results of parameter sensitivity

build-log-based query augmentation, which is based on the text similarity between the inconsistent artifact names and the build commands, and using this text similarity tends to be less accurate.

In addiction, an interesting observation is over the variants without the DID mechanism. For example, despite not performing well in RQ1, REPTRACE(¬DID) achieves an *R@10* of 0.7029 in this RQ. A similar phenomenon could be found for REPTRACE(¬DID, ¬RID) as well. The reason might be that the retrieved build commands by these two variants may still be relevant to the inconsistent artifacts, even when they are not the root causes for unreproducibility. Hence, these build commands may be helpful in file-level localization.

Similar to RQ1, we present the results of hypothesis testing in Table VI. The table is organized the same way as RQ1. From the table, similar phenomena could be observed. Moreover, when comparing the topmost retrieved files by REPTRACE with REPLOC, we find that REPTRACE outperforms REPLOC, except that the p-value is slightly larger than 0.05 when the recall metric is considered.

Answer to RQ4: REPTRACE is able to accurately locate the problematic files that are responsible for the unreproducible builds. From the comparisons with both the state-of-the-art approach and the variants of REPTRACE, REPTRACE demonstrates the superiority over these approaches.

V. THREATS TO VALIDITY

In our evaluation, there are two major threats to validity.

First, an important threat to validity is that we assume the completeness of the necessary system call traces, which may introduce inconsistencies during the build process. For example, in our evaluation, all the builds are conducted under an isolated environment, and do not need to communicate with external systems once the build dependencies are met. Hence, we do not capture the network-related system calls. In real-world environments, inconsistencies could originate from various sources. Hence, the linkage from the inconsistent artifacts toward the root cause may be broken. During the construction of the dataset, we have mitigated this issue by manually inspecting the patches and the build scripts, to ensure that the unreproducible issues are caused within the package.

Second, in our evaluation we adopt the off-the-shelf diagnosis tool ***strace*** to capture the system call traces. ***strace*** is based on ***ptrace***, and is available under GNU/Linux. To generalize REPTRACE to other platforms, adaptations have to

TABLE V
RESULTS OF REPTRACE AND OTHER APPROACHES FOR THE FILE-LEVEL LOCALIZATION TASK

Approach	*A@1*	*A@5*	*A@10*	*P@1*	*P@5*	*P@10*	*R@1*	*R@5*	*R@10*	*F-1@1*	*F-1@5*	*F-1@10*	*MRR*
REPTRACE(¬DID, ¬RID)	0.3889	0.7222	0.8000	0.3889	0.1667	0.0933	0.3403	0.6765	0.7598	0.3559	0.2606	0.1634	0.5143
REPTRACE(¬DID)	0.2556	0.6778	0.7333	0.2556	0.1567	0.0856	0.2139	0.6445	0.7029	0.2278	0.2462	0.1501	0.4316
REPTRACE(¬RID)	0.6444	0.8556	0.8611	0.6444	0.1967	0.1011	0.5727	0.8149	0.8251	0.5957	0.3098	0.1772	0.7320
REPLOC	0.5611	0.8333	0.8722	0.5611	0.1878	0.1017	0.5006	0.7839	0.8343	0.5189	0.2958	0.1780	0.6815
REPTRACE	0.6667	0.8778	0.9056	0.6667	0.2000	0.1067	0.5835	0.8311	0.8695	0.6091	0.3152	0.1868	0.7583

TABLE VI
RESULTS OF WILCOXON SIGNED RANK TEST FOR THE FILE-LEVEL LOCALIZATION TASK

Metrics	REPTRACE *vs.*	*p*-value	Effect size
P@1	REPTRACE(¬DID,¬RID)	<0.0001	0.7353
	REPTRACE(¬DID)	<0.0001	0.8222
	REPTRACE(¬RID)	0.3711	0.2000
	REPLOC	0.0184	0.2923
R@1	REPTRACE(¬DID,¬RID)	<0.0001	0.7131
	REPTRACE(¬DID)	<0.0001	0.8139
	REPTRACE(¬RID)	0.7168	0.0857
	REPLOC	0.0655	0.2485

be made. To mitigate this issue, we model the system calls in a uniform way (see Section II-B), so that porting to other platforms would be straightforward. The adaptation could be realized by replacing ***strace*** with a platform-specific tracing system, e.g., ***DTrace*** [19] for BSD-like OS and ***ETW*** [20] for Windows.

VI. RELATED WORK

A. System Calls

Recent years have witnessed the growing research interests of leveraging system call traces as a high-quality source of system-wide information, to help boost the performance of various tasks. For instance, Gao et al. [21] propose to use system calls to capture the trajectories of malware behaviors [22], which could be further used to detect intrusion, or conduct forensic analysis. Licker and Rice [23] address the challenge of discovering the hidden dependency in the build scripts, and detect bugs in the build process. Neves et al. [24] develop a system-call-tracing-based diagnosis framework, Falcon, to achieve trouble-shooting functionality under distributed environments. Pasquier et al. [25] propose a whole-system provenance system that leverage system call to capture meaningful provenance without modifying existing applications. Van Der Burg et al. [26] address the license compatibility problem, and devise a system-call-based approach to detect potential license conflict. Liu et al. [27] systematically review the studies focusing on host-based intrusion detection with system calls.

Unlike the existing system-call-based research, in this work, we focus on a novel problem domain, i.e., the localization task of the root causes for unreproducible builds.

B. Reproducibility

As a new research problem, there are relatively few approaches focusing on the localization task for unreproducible builds. The most relevant work is the work by Ren et al. [7], in which the localization task is modeled as a task of information retrieval, aiming to search for the problematic files that are responsible for the unreproducibility. Also, in their work, the localization is realized at the file level, unlike the level of build command achieved in this work.

Besides the localization task for unreproducible builds, there exist a series of closely related research directions. Among these directions, a typical example is reproducible research. For example, Guo [28] proposes a system-call-based framework, CDE, which realizes the functionality of packaging the program-execution environment. Following the idea, there exist several related approaches, such as ReproZip [11] and ProvToolbox [29]. Ivie and Thain [30] make a systematic survey for the research topic. Compared with these previous approaches, which emphasize the success of re-executing the programs in diverse environments, in this work we are more interested in tracing back along the system calls, to locate the root cause for inconsistencies.

VII. CONCLUSION

In this paper, we have presented the REPTRACE framework to identify the root causes for unreproducible builds. The framework leverages system call tracing's uniform interfaces for monitoring executed build commands in diverse build environments. To tackle the challenges of leveraging system-call-tracing-based information, REPTRACE filters irrelevant dependencies among processes by using the differences of the write data and the runtime values. Our extensive evaluation over real-world packages demonstrates that REPTRACE is able to achieve promising solutions for unreproducible builds.

In future work, as REPTRACE relies on the heuristic detection of the dependencies between parent processes and child processes, we plan to explore more accurate techniques for dependency identification. Also, it would be interesting to explore the possibility of automatically patching unreproducible builds.

ACKNOWLEDGMENTS

Tao Xie is also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. This work is supported in part by the National Key Research and Development Program of China under grant no. 2018YF-B1003900, the National Natural Science Foundation of China under grant no. 61772107, 61722202, 61529201, and NSF under grant no. CNS-1564274, CCF-1816615, CNS-1755772.

REFERENCES

[1] Reproducible builds team, "Definition of reproducible builds," https://reproducible-builds.org/docs/definition/, 2018, accessed: 2019-03-04.
[2] M. Perry, "Deterministic builds part one: Cyberwar and global compromise," https://blog.torproject.org/deterministic-builds-part-one-cyberwar-and-global-compromise, 2013, accessed: 2019-03-04.

[3] B. Bzeznik, O. Henriot, V. Reis, O. Richard, and L. Tavard, "Nix as HPC package management system," in *Proceedings of the Fourth International Workshop on HPC User Support Tools*. ACM, 2017, pp. 4:1–4:6.
[4] "Debian," https://www.debian.org/, accessed: 2019-03-04.
[5] "Guix," https://www.gnu.org/software/guix/, accessed: 2019-03-04.
[6] "F-Droid," 2019, https://f-droid.org/.
[7] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, "Automated localization for unreproducible builds," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 71–81.
[8] 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," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1587–1633, 2015.
[9] Y. Régis-Gianas, N. Jeannerod, and R. Treinen, "Morbig: A static parser for POSIX shell," in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2018, pp. 29–41.
[10] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li, "NodeMerge: Template based efficient data reduction for big-data causality analysis," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1324–1337.
[11] F. Chirigati, D. Shasha, and J. Freire, "ReproZip: Using provenance to support computational reproducibility," in *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, 2013.
[12] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. ACM, 2015, pp. 184–197.
[13] B. Zong, X. Xiao, Z. Li, Z. Wu, Z. Qian, X. Yan, A. K. Singh, and G. Jiang, "Behavior query discovery in system-generated temporal graphs," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 240–251, Dec. 2015.
[14] "Strace," https://strace.io, accessed: 2019-03-04.
[15] "Cosine similarity," https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/CosineSimilarity.html, accessed: 2019-03-04.
[16] "Longest common substring percentage," https://www.oracle.com/webfolder/technetwork/data-quality/edqhelp/Content/processor_library/matching/comparisons/longest_common_substring_percentage.htm, accessed: 2019-03-04.
[17] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 689–699.
[18] D. S. Kerby, "The simple difference formula: An approach to teaching nonparametric correlation," *Comprehensive Psychology*, vol. 3, p. 11.IT.3.1, 2014.
[19] "DTrace," https://dtrace.org, accessed: 2019-03-04.
[20] M. Jacobs and M. Satran, "About event tracing," https://docs.microsoft.com/en-us/windows/desktop/etw/about-event-tracing, accessed: 2019-03-04.
[21] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling efficient attack investigation from system monitoring data," in *Proceedings of 2018 USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 113–126.
[22] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 2018, pp. 639–656.
[23] N. Licker and A. Rice, "Detecting incorrect build rules," in *41st ACM/IEEE International Conference on Software Engineering*. ACM/IEEE, 2019, pp. 1234–1244.
[24] F. Neves, N. Machado, and J. Pereira, "Falcon: A practical log-based analysis tool for distributed systems," in *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE/IFIP, 2018, pp. 534–541.
[25] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 405–418.
[26] S. Van Der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel, "Tracing software build processes to uncover license compliance inconsistencies," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM/IEEE, 2014, pp. 731–742.
[27] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Computing Surveys*, vol. 51, no. 5, p. 98, 2018.
[28] P. J. Guo, "CDE: Run any linux application on-demand without installation," in *Proceedings of the 25th USENIX International Conference on Large Installation System Administration*. USENIX Association, 2011, pp. 2–2.
[29] L. Moreau, B. V. Batlajery, T. D. Huynh, D. Michaelides, and H. Packer, "A templating system to generate provenance," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 103–121, 2018.
[30] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Computing Surveys*, vol. 51, no. 3, p. 63, 2018.