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 REP TRACE, 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, REP TRACE 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, REP TRACE 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 REP TRACE over a set of real-world software packages show that REP TRACE 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, REP TRACE 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 REP TRACE over a set of real-world software packages show that REP TRACE 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, REP TRACE 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].

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 metadata1, 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 reprotest2 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

1https://tracker.debian.org/pkg/disorderfs
2https://tracker.debian.org/pkg/reprotest
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, REP TRACE, 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 ac-
curate 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,
REP TRACE builds a dependency graph of inconsistent arti-
facts based on two types of dependencies, and searches the
graph to identify the process that causes the inconsistencies.
Specifically, REP TRACE 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,
REP TRACE starts from the inconsistent artifact, and then iden-

\(^3\)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 B1 and B2, respectively. Each round of build comprises a sequence of processes, e.g., \{P1, P2, P3, \ldots\}. Specifically, a process is represented as a tuple \((PID, PPID, slist)\), where the first two fields are self-explanatory, and the slist field indicates a list of system call traces. Each system call \(s \in slist\) is represented by a tuple \((type, start-time, end-time, source, target, data)\). 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 ending time of the system call. The remaining fields are system call specific:

1) read represents the read system call and its variants, such as \texttt{read} and \texttt{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 \texttt{write} and \texttt{pwritev}. The target field specifies the file to write, and the data indicates the bytes written to target.
3) rename represents the system calls of rename, 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 \(V = \{s.data_s \in P.slist, s.type = read, write, or execve\}\). 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 \((PID_1, PPID_1, slist_1)\) and \((PID_2, PPID_2, slist_2)\) of the same build, a read/write dependency \((PID_1 \rightarrow PID_2)\) is established if (1) \(\exists (read, s_1, et_1, f_s, data_s) \in slist_1, (write, s_2, et_2, f_s, data_s) \in slist_2, s\) such that \(et_1 > et_2\), or if (2) \(\exists (read, s_1, et_1, f_s, data_s) \in slist_1, (write, s_2, et_2, f_s, data_s) \in slist_2, s\) such that \(et_1 > et_3 > et_2\).

Definition 3 (parent/child process dependency): Given two processes \(P_1\) and \(P_2\) with PIDFs \(p_1\) and \(p_2\), respectively, of the same round of build, if \(p_1.PPID = P_2.PID\), 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 file to write, and the data indicates the bytes written to target.

Fig. 1. System call trace snippet for \textit{airstrike}

1 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.

529
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 the two 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 → 4212). Following this clue, we could traverse to the process with PID 4208 (cc). At this point, the hint for further traversal (4208 → 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 → 4002), where $f$ is a file (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, REP TRACE 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 REP TRACE FRAMEWORK

In this section, we describe the design and implementation of the proposed REP TRACE 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.

<table>
<thead>
<tr>
<th>PID</th>
<th>Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>4420</td>
<td>objcopy […] debian/airstrike/usr/games/airstrike</td>
</tr>
<tr>
<td>4419</td>
<td>strip […] debian/airstrike/usr/games/airstrike</td>
</tr>
<tr>
<td>4242</td>
<td>cp -reflink=auto -a, debian/tmp/usr/games, debian/airstrike/usr/airstrike</td>
</tr>
<tr>
<td>4240</td>
<td>install airstrike […] /usr/games/airstrike</td>
</tr>
<tr>
<td>4213</td>
<td>ld […] -&gt; airstrike […] /players.o /airedrke.o […]</td>
</tr>
<tr>
<td>4021</td>
<td>as -l […] -o airstrike.o /tmp/cc3w6oB6L.s</td>
</tr>
<tr>
<td>4142</td>
<td>as -l […] -o airstrike.o /tmp/cc3b3bOBX.s</td>
</tr>
<tr>
<td>4212</td>
<td>collect2 […] -&gt; airstrike […] /players.o /airedrke.o […]</td>
</tr>
<tr>
<td>4208</td>
<td>sh -c cc -&gt; airstrike […] /players.o /airedrke.o […]</td>
</tr>
<tr>
<td>4000</td>
<td>make -C src airstrike</td>
</tr>
<tr>
<td>4002</td>
<td>find . -name *.c</td>
</tr>
</tbody>
</table>

Table II: INCONSISTENT ARTIFACTS AND TYPICAL BUILD COMMANDS FOR THE DEPENDENCY GRAPH OF airstrike.

Fig. 2. Dependency graph for airstrike.
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 system call traces, 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 \( \bigcup_{P \in B_1} \bigcup_{s.cP \in \text{slist}} \text{md5}(s.data) \) — \( \bigcup_{P \in B_2} \bigcup_{s.cP \in \text{slist}} \text{md5}(s.data) \). Given two processes \( \langle PID_1, PPID_1, \text{slist}_1 \rangle \in B_1 \), \( \langle PID_2, PPID_2, \text{slist}_2 \rangle \in B_1 \), a difference-induced dependency (DID, denoted as \( PID_1 \xrightleftharpoons{\text{write-diff}} PID_2 \)) is established, if \( PID_1 \xrightarrow{\text{write-diff}} PID_2 \), and \( \exists \{\text{write, st, et, f, data}\} \in \text{slist}_2 \), such that \( \text{md5}(\text{data}) \in \text{write-diff} \).

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 \( s_3 \), 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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( G \leftarrow \emptyset \text{graph} )</td>
</tr>
<tr>
<td>2</td>
<td>( \text{write-hash}<em>1 \leftarrow \bigcup</em>{P \in B_1} \bigcup_{s.cP \in \text{slist}} \text{md5}(s.data) )</td>
</tr>
<tr>
<td>3</td>
<td>( \text{write-hash}<em>2 \leftarrow \bigcup</em>{P \in B_2} \bigcup_{s.cP \in \text{slist}} \text{md5}(s.data) )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{write-diff} \leftarrow \text{write-hash}_1 \xrightarrow{\text{write-diff}} \text{write-hash}_2 )</td>
</tr>
<tr>
<td>5</td>
<td>for write system call ( s_1 ) where ( \text{md5}(s_1) \in \text{write-diff} ) do</td>
</tr>
<tr>
<td>6</td>
<td>( \text{pid-write} \leftarrow \text{pid-off}(s_1) )</td>
</tr>
<tr>
<td>7</td>
<td>for read system call ( s_2 ) do</td>
</tr>
<tr>
<td>8</td>
<td>if ( s_2 ), src ( = s_1 ), dest and ( s_2 ), et ( &gt; s_1 ), et then</td>
</tr>
<tr>
<td>9</td>
<td>( \text{add-edge}(G, \text{pid-read}, \text{pid-write}) )</td>
</tr>
<tr>
<td>10</td>
<td>end</td>
</tr>
<tr>
<td>11</td>
<td>end</td>
</tr>
<tr>
<td>12</td>
<td>end</td>
</tr>
<tr>
<td>13</td>
<td>for rename system calls ( s_3 ) do</td>
</tr>
<tr>
<td>14</td>
<td>if ( s_3 ), src ( = s_1 ), dest and ( s_3 ), dest ( = s_2 ), src and</td>
</tr>
<tr>
<td>15</td>
<td>( s_3 ), et ( &gt; s_2 ), et and ( s_2 ), et ( &gt; s_1 ), et then</td>
</tr>
<tr>
<td>16</td>
<td>( \text{add-edge}(G, \text{pid-read} \rightarrow \text{pid-off}(s_2)) )</td>
</tr>
<tr>
<td>17</td>
<td>end</td>
</tr>
<tr>
<td>18</td>
<td>end</td>
</tr>
<tr>
<td>19</td>
<td>end</td>
</tr>
<tr>
<td>20</td>
<td>return ( G )</td>
</tr>
</tbody>
</table>

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 \( p_1 \) and \( p_2 \), each with a sequence of runtime values \( V_1 = \{v_1, v_2, \ldots, v_{m_1}\} \) and \( V_2 = \{v_{21}, v_{22}, \ldots, v_{2n}\} \), the relevance between the two processes is calculated as

\[
\text{relevance}(p_1, p_2) = \max_{v_1 \in V_1, v_2 \in V_2} \left\{ \max\{\cosSim(v_1, v_2), \text{lcsSim}(v_1, v_2)\} \right\},
\]

where \( \cosSim \) and \( \text{lcsSim} \) represent the cosine-based [15] and the longest-common-substring-based [16] similarity, respectively. Note that for \( \text{lcsSim} \), 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 \( p_1 \) and \( p_2 \) of the same round of build, there exists a runtime-value-induced dependency (RID, denoted as \( p_1 \xleftarrow{\text{runtime}} p_2 \)) if \( p_1 \Rightarrow p_2 \), and the relevance value between the processes is larger than the pre-defined threshold.
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.

• 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,
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, REP TRACE 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 REP TRACE on real-world software packages and evaluate the effectiveness of REP TRACE. We seek to investigate the following research questions (RQs):

1) **RQ1:** Is REP TRACE 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 REP TRACE sensitive to the parameter in the runtime-value-induced dependency?
4) **RQ4:** Is REP TRACE 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 REP TRACE. In particular, by comparing various variants of REP TRACE, we intend to examine how each component contributes to REP TRACE. 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 REP TRACE, we could gain more insights into both the DID and RID mechanisms. RQ3 evaluates the sensitivity of the parameter on REP TRACE. Finally, RQ4 evaluates the ability of file-level localization of REP TRACE by comparing with the best known results.

**A. Evaluation Setup**

REP TRACE 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 REP TRACE, 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 REP TRACE. 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 \( \text{Prec@N}, \text{Rec@N}, \text{F-1@N} \)). In particular, Top-N accuracy rate, e.g., \( \text{A@N} \), is used to measure the accuracy rate of the Top-N list for 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 REP TRACE with a set of variants. First, a set of three variants of REP TRACE are chosen, each considering part of the mechanisms of REP TRACE. For example, we denote REP TRACE\((\neg \text{DID})\) as the variant of REP TRACE in which the difference-induced dependency is not employed. There are also two other variants denoted as REP TRACE\((\neg \text{RID})\) and REP TRACE\((\neg \text{DID}, \neg \text{RID})\), 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 REP TRACE 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
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 REP TRACE 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 REP TRACE lies in its ability to locate the root causes for unreproducible builds. Prior to this work, the localization task realized by REP LOC [7] is mainly at the file level. Hence, the guidance toward the patch of the unreproducible builds tends to be limited. In contrast, with the system call tracing, especially the data provided by the `execve` system call, REP TRACE 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 REP TRACE 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 REP TRACE is able to effectively locate the root causes responsible for the un reproducibility. Especially, when considering the topmost retrieved build command, REP TRACE 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 REP TRACE 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 un reproducibility, 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 REP TRACE 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 REP TRACE 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 REP TRACE with REP TRACE (~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 REP TRACE 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 REP TRACE (~RID) with REP TRACE (~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 accurately. A similar observation could be made when we compare REP TRACE (~RID) and REP TRACE. Furthermore, to understand whether the RID mechanism works, we compare REP TRACE with REP TRACE (~RID). From Fig. 6, we could see that REP TRACE outperforms REP TRACE (~RID).

**Answer to RQ1:** REP TRACE is able to effectively identify
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
Answer to RQ3: REP TRACE 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 REP TRACE is effective in locating the problematic file, in which the unreproducible issues should be patched. Specifically, we report the results obtained by REP TRACE and other approaches under comparison in Table V. The table is organized similarly as Table III, except that REP LOC is also considered.

From the table, we can observe that REP TRACE is able to rank the relevant files at the top of the retrieved list. Compared with REP LOC, the Top-1 accuracy rate is 0.6667, which is much higher than the results achieved by REP LOC. The underlying reason might be that with the system-call-based dependency graph, REP TRACE 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, REP LOC relies on the 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 addition, an interesting observation is over the variants without the DID mechanism. For example, despite not performing well in RQ1, REP TRACE (¬DID) achieves an R@10 of 0.7029 in this RQ. A similar phenomenon could be found for REP TRACE (¬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 REP TRACE with REP LOC, we find that REP TRACE outperforms REP LOC, except that the p-value is slightly larger than 0.05 when the recall metric is considered.

Answer to RQ4: REP TRACE 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 REP TRACE, REP TRACE 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 REP TRACE to other platforms, adaptations have to...
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 `trace` 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 REP TRACE 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, REP TRACE 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 REP TRACE is able to achieve promising solutions for unreproducible builds.

In future work, as REP TRACE 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

HPC package management system,” in Proceedings of the Fourth
unreproducible builds,” in Proceedings of the 40th International
large-scale empirical study of the relationship between build technology
and build maintenance,” Empirical Software Engineering, vol. 20, no. 6,
parser for POSIX shell,” in Proceedings of the 11th ACM SIGPLAN
International Conference on Software Language Engineering. ACM,
2018, pp. 29–41.
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.
to support computational reproducibility,” in Proceedings of the 5th
causal profiling,” in Proceedings of the 25th ACM Symposium on
and G. Jiang, “Behavior query discovery in system-generated temporal
webfolder/technetwork/data-quality/edqhelp/Content/processor_library/
matching/compare/longest_common_substring_percentage.htm,