Automated Patching for Unreproducible Builds

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

Shiwei Sun
School of Software, Dalian University of Technology
Dalian, China
21917046@mail.dlut.edu.cn

Jifeng Xuan
School of Computer Science, Wuhan University
Wuhan, China
jxuan@whu.edu.cn

Xiaochen Li
University of Luxembourg
Luxembourg
School of Software, Dalian University of Technology
Dalian, China
xiaochen.li@uni.lu

Zhide Zhou
School of Software, Dalian University of Technology
Dalian, China
cszide@gmail.com

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

ABSTRACT

Software reproducibility plays an essential role in establishing trust between source code and the built artifacts, by comparing compilation outputs acquired from independent users. Although the testing for unreproducible builds could be automated, fixing unreproducible build issues poses a set of challenges within the reproducible builds practice, among which we consider the localization granularity and the historical knowledge utilization as the most significant ones. To tackle these challenges, we propose a novel approach RepFix that combines tracing-based fine-grained localization with history-based patch generation mechanisms.

On the one hand, to tackle the localization granularity challenge, we adopt system-level dynamic tracing to capture both the system call traces and user-space function call information. By integrating the kernel probes and user-space probes, we could determine the location of each executed build command more accurately. On the other hand, to tackle the historical knowledge utilization challenge, we design a similarity based relevant patch retrieving mechanism, and generate patches by applying the edit operations of the existing patches. With the abundant patches accumulated by the reproducible builds practice, we could generate patches to fix the unreproducible builds automatically.

To evaluate the usefulness of RepFix, extensive experiments are conducted over a dataset with 116 real-world packages. Based on RepFix, we successfully fix the unreproducible build issues for 64 packages. Moreover, we apply RepFix to the Arch Linux packages, and successfully fix four packages. Two patches have been accepted by the repository, and there is one package for which the patch is pushed and accepted by its upstream repository, so that the fixing could be helpful for other downstream repositories.

*corresponding author

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510102

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software testing and debugging**.

KEYWORDS

reproducible builds, dynamic tracing, automated patch generation

ACM Reference Format:
Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. 2022. Automated Patching for Unreproducible Builds. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510102

1 INTRODUCTION

As a set of emerging software engineering practices, reproducible builds have attracted rapidly growing interests from both academia and industry. The motivation behind reproducible builds is to allow any user to verify that no vulnerabilities or backdoors have been introduced during the compilation process. Through validation, localization, and repairing tasks, the reproducible builds aim at building bit-for-bit identical compiled packages, to bridge the gap between source to binary code with an independently-verifiable path[28]. For example, the well-known malware XcodeGhost, which affected more than 4,000 packages, could be detected by independent recompiling applications from multiple build environments[18]. By guaranteeing identical built artifacts are always generated from a given source package, diverse third party users could come to a consensus on the build result, so that inconsistent built artifacts immediately trigger alarms for further investigation.

Within the reproducible builds practice, fixing unreproducible build issues is an important meanwhile challenging task. Currently, various software repositories are conducting the testing, the localization, and the fixing for the package reproducibility issues. For example, industry-leading companies such as Microsoft, Google, and Huawei have been focusing on the reproducibility property of their software product lines[14, 26, 39]. In the open source community, GNU/Linux distributions such as Debian, Arch Linux, and Guix are routinely validating the reproducibility of the packages hosted by their repositories[1]. As of February 2022, 92.9% of Debian's packages (bookwarm/amd64) and 78.8% of Arch Linux's packages are reported to be reproducible[2]. Also, studies focusing on the localization task, i.e., searching for the root causes and problematic files for unreproducible builds, have been reported in the literature[37, 38].

While the automation of validation and localization has been developed as emerging techniques, the fixing for the unreproducible builds is mostly manually conducted, and relies heavily on the developers' knowledge and experience. There exist various obstacles within the automated fixing process for the unreproducible builds, and we list two major technical challenges as follows.

- **Localization granularity challenge**. Despite the promising results achieved [37, 38], the localization for unreproducible builds could only be realized at the file-level, i.e., developers have to manually read the source file reported by the localization tools, in search of the specific line for patching. Given that the Makefiles and scripts could be of tens to hundreds of lines, such fine-grained localization task could be time-consuming and error-prone.
- **Historical knowledge utilization challenge**. Currently, the patches for fixing unreproducible build issues are mostly manually written by the developers. Meanwhile, for software distributions like Debian, during the reproducible builds practices, various patches for fixing the unreproducible build issues have been accumulated. However, how to leverage the historical knowledge to fix new unreproducible packages remains a great challenge.

To tackle the aforementioned challenges, we propose a novel Reproducible build Fixing (RepFix) approach, which features the combination of two mechanisms, i.e., the tracing-based fine-grained localization and the history-based patch generation. On the one hand, to face the localization granularity challenge, we incorporate not only kernel-level system call traces, but also user-space application traces to establish the linkage between each build command and its specific invocation location. With the help of these runtime traces, problematic build commands with their accurate location could be located. On the other hand, to tackle the historical knowledge utilization challenge, we propose a patch generation approach guided by the existing patches. For unreproducible packages, we retrieve their most relevant patches, extract the edit operations from the retrieved patches, and apply the operations over the problematic build command obtained from the fine-grained localization. With the patched source files, we are able to validate the correctness of the overall approach.

To evaluate RepFix, we take the real-world packages from Debian as a case study, to examine whether RepFix is able to generate valid patches that fix unreproducible build issues. Over the 116 packages, RepFix is able to successfully fix all the unreproducible build issues over 33 packages, and partially fix the issues over another 31 packages. Moreover, to examine the generalization of RepFix, we apply RepFix over the Arch Linux packages, and successfully fix the unreproducible build issues for four packages, of which two patches have been accepted. In particular, there is one package for which the patch is pushed and accepted by its upstream repository. We make the details of the patches available at https://rezilla.bitbucket.io/repfix.

The contributions of this study could be summarized as follows:

- To the best of our knowledge, we are the first to generate patches for unreproducible builds in an automated paradigm.
- We propose a tracing-based approach that unifies traces from kernel and user-space to realize fine-grained localization, and design a history-based patch generation.
- We conduct extensive experiments over the dataset collected from a set of real-world unreproducible packages, to demonstrate the effectiveness of RepFix. We also submit four patches constructed by RepFix to the Arch Linux bug tracking system, among which two have been accepted by the maintainers.

The remainder of this paper is organized as follows. In Section 2, we introduce the background information with a motivating example. In Section 3, we discuss the details of the RepFix approach. In Section 4, extensive experiments are conducted to evaluate RepFix from various perspectives. Sections 5 – 6 present the discussion, as well as the related work of this study. Finally, Section 7 concludes this study, and points out the future research directions.

2 MOTIVATING EXAMPLE

In this section, we introduce the background information with a motivating example. Take the *mylvmbackup* package, a MySQL backup utility (with version 0.15-1) from the Debian repository as an example[9], we first describe the reproducibility validation workflow. The validation process is carried out by building the source files under controlled, varied build configurations. The altered configurations include build date, timezone information, locale, file system traversing order, etc[3].

If the built artifacts under the two build configurations are bit-for-bit identical, the package is reported as reproducible. Otherwise, if there exists any inconsistent artifact between the two builds, the package is indicated as unreproducible, and we shall continue to analyze the root cause for the unreproducible build issue, conduct the localization task, and fix the problematic build commands. With the reproducibility validation tool chain *reprotest*[4], *mylvmbackup* is reported as unreproducible. In Fig. 1, we present the diff log for the package, which is generated by the in-depth comparison utility *diffoscope*[5]. It is shown that there exists an inconsistent artifact `/usr/bin/mylvmbackup`, in which a timestamp is embedded in the generated file. Consequently, when the package is built at different time, inconsistent packages will be compiled.

To fix unreproducible build issues, localization has to be first conducted. Currently there exist automatic approaches such as RepLoc

[1] https://reproducible-builds.org/projects/
[2] https://reproducible-builds.org/citests/
[3] https://reproducible-builds.org/docs/perimeter/
[4] https://pypi.org/project/reprotest/
[5] https://diffoscope.org

```
"source1": "./usr/bin/mylvmbackup",
"source2": "./usr/bin/mylvmbackup",
unified_diff": "@@ -31,15 +31,15 @@
 use Fcntl;

 use diagnostics;
 use strict;

 # Version is set from the Makefile
 my $version='0.15';
-my $build_date='2021-08-17';
+my $build_date='2022-09-04';
...
```

Figure 1: Diff log for mylvmbackup

```
38 # define some variables
39
40 NAME = mylvmbackup
41 VERSION = 0.15
42 BUILDDATE = $(shell date +%Y-%m-%d)
43 MAN1 = man/$(NAME).1
44 HOOKS := $(wildcard hooks/*.pm)
45 DISTFILES = \
46         ChangeLog \
47         COPYING \
48         CREDITS \
...
```

Figure 2: Snippet of the `/Makefile` file for *mylvmbackup*

[37] and RepTrace [38], which aim at retrieving the problematic files that cause the unreproducible build issues. For the two tools, RepLoc follows the information retrieval based fault localization studies[29, 44], and realizes the localization functionality based on text similarity between inconsistent artifacts and build logs, to search for the most relevant build scripts to the inconsistent artifacts. Meanwhile, besides file-level localization, RepTrace is able to represent the root cause as the problematic build command and its process ID (pid). For both the approaches, the `/Makefile` could be located. However, the file has to be further manually traversed, to identify the 42th line that should be patched (see Fig. 2). After that, the line of the problematic build command should be patched, by modifying the command, in the hope of fixing the issue.

Ideally, if we could obtain the mapping between each executed command and its location where the command is invoked, it will be helpful for fixing the issue. However, obtaining such mapping relationship is not straightforward. A possible way is to instrument the Bash interpreter, adding logging statements in the source code. However, such intrusive approach is hard to generalize to other applications. Alternatively, another possible way is to leverage the power of dynamic tracing frameworks such as SystemTap[6] and bpftrace[7]. These frameworks allow developers to deeply investigate the behavior of the kernel and user-space applications, in order to debugging errors[16, 25], performance issues[19], or understand system working mechanisms[31]. The main focus of these tools is to make it easy to capture and manipulate the required data without modifying the kernel/application source code, which is often required for instrumentation-based studies[15, 27]. Both SystemTap and bpftrace are command line applications that utilize scripts as input and generates plain text output. The expressiveness of the domain-specific tracing languages makes it possible to generalize to other applications. Based on the connection between the traces from user-space and kernel-space, we could extend the causality analysis to achieve fine-grained localization.

Besides, since our goal is to generate feasible patches, we intend to design an extensible approach to automate such process. Currently, the patches are manually constructed, based on the developers' experience. Meanwhile, during the reproducible builds practice, software repositories like Debian have accumulated thousands of patches for fixing unreproducible build issues. Hence, an automated approach based on the historically fixed patches would be ideal. However, the cumulated knowledge may not be directly transferable to new unreproducible packages. For example, to guide the patch generation, we have to take the grammar of the build scripts into consideration.

3 PROPOSED APPROACH

In this section, we discuss the design and implementation of the RepFix framework. In Fig. 3, we first illustrate the components of the proposed framework. In RepFix, there are two major components, i.e., tracing-based fine-grained localization and history-based patch generation, which aim to tackle the localization granularity challenge and the historical knowledge utilization challenge, respectively.

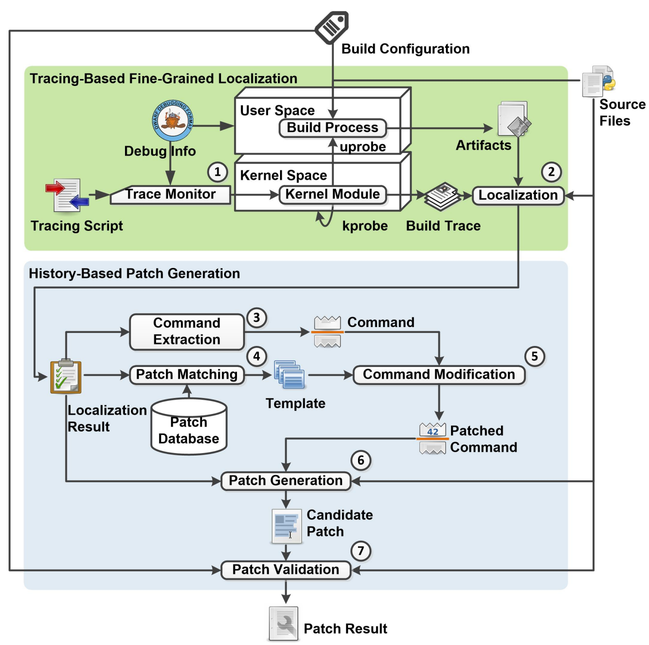


Figure 3: The RepFix framework

More specifically, in the fine-grained localization component, we first invoke the build process under the supervision of the tracing monitor (step 1). The tracing script defines a set of probes, which could be attached to both the kernel (kprobes) and user-space applications (uprobes). With these probes, runtime traces such as the

[6]https://sourceware.org/systemtap/
[7]https://github.com/iovisor/bpftrace

parameters, return value, and global/local variables of system calls and user-space functions could be collected during the build process. After the build completes, the localization process is launched, with the traces, the source files, and the built artifacts as the input (step 2).

After the localization, history-based patch generation is conducted. We extract the command to patch from the localization result (step 3), and get the most relevant patches from Debian's bug tracking system (step 4). Under the guidance of the retrieved patch, the command is modified (step 5), and further used to generate the candidate patch (step 6). With the generated patch, reproducibility validation should be applied over the patched source files, to evaluate the patch (step 7). Finally, if the validation succeeds, the patch is returned for deeper investigation.

In the subsequent subsections, we shall discuss the two components in more details.

Algorithm 1: Tracing-based Fine-grained Localization

Input: Source files *src*, Build configuration *conf*
Output: Localization result *res*

```
1  begin
2      for i ∈ {1, 2} do
3          ktrace_i, utrace_i ← build(src, conf_i)
4      end
5      pid_list ← RepTrace (ktrace)
6      lmap ← location-map(utrace)
7      res ← list()
8      for pid ∈ pid_list do
9          res ← append(res, lmap[pid])
10     end
11     return res
12 end
```

3.1 Tracing-Based Fine-Grained Localization

In the tracing-based localization component, we build the source files twice to obtain the built artifacts. In particular, to gain deep observability of the build process, a trace monitor is employed to capture both the system call information and the user-space process runtime information. In this study, we adopt SystemTap to realize the tracing functionality, due to its expressiveness and efficiency. An advantage of using SystemTap in our approach lies in its ability to capture both kernel-space and user-space traces. Hence, after the build process, we could not only capture what build commands have been executed as in the existing studies [36, 38], but also where these commands are invoked.

On the one hand, for the system call traces, kprobes are defined in the script, which are translated and compiled into kernel modules. On the other hand, with the help of DWARF (Debugging With Attributed Record Formats)[8] debug information, uprobes could also be defined. In this study, we consider the user-space runtime traces of two types of build scripts, i.e., Bash and Make from the GNU project. The reasons we consider these two types of scripts are as follows. First, both Bash and Make are among the most popular build tools, which are widely used in the open source community [21, 35]. Second, both the tools are highly dynamic. For instance, according to the maintainer of GNU Make, there is no official grammar for Make, since Makefiles could be highly context-dependant [40]. Hence, it is difficult to implement static analysis for these build tools, especially for those scenarios where multiple build tools are involved. As a result, it is reasonable to take Bash and Make as the case study, to investigate the feasibility of realizing localization at line-level.

For Bash, the runtime traces could be obtained by probing the `make_child` user-space function. Each time a command in a Bash script is invoked, the `make_child` function will be called, and the return value of the function indicates the pid of the executed command. Moreover, the source file and line number of the command could be extracted from global variables `shell_script_filename` and `currently_executing_command`. By attaching probes to the `make_child` function of the Bash executable, we are able to bridge the gap between the pid of each command and its corresponding location. Similarly, for Make, the runtime traces could be extracted from two user-space functions, i.e., `start_waiting_job` and `job_next_command`. For both functions, the parameter refers to an instance of `struct child`, which encapsulates the fields such as `filenm` and `lineno`. Besides, we also attach probes to the function `lookup_variable`, to capture the locations of variable definitions in Makefiles. With such information, we are able to complete the localization task, by establish linkages between the pid obtained from RepTrace and the line-level location for patch. It is interesting that being a byproduct in RepTrace, the pid plays an essential role in connecting the high-level localization based on system call tracing and the low-level, fine-grained localization based on user-space function call tracing.

In Algo. 1, we present the pseudo code of the tracing-based fine-grained localization. The localization is based on the system-call based localization as in RepTrace. First, we build the source files twice with varied configurations (lines 1–4). Meanwhile, we apply SystemTap to capture the traces from kernel and the build tools (Bash and Make), which are indicated as the *ktrace* and the *utrace*. On the one hand, with *ktrace*, we are able to apply RepTrace to locate the problematic build command, with their corresponding pid (line 5). On the other hand, based on *utrace*, we could construct a key-value structure *lmap* (line 6), with which we are able to query the location with the pid of the build command. Hence, we could transfer the results of RepTrace into the location for patching (lines 7–10).

Running Example: Consider the *mylvmbackup* package introduced in Section 2, Fig. 4 illustrates the overall workflow of the localization procedure. First, the kprobes and uprobes are attached to the kernel and the build tools, i.e., Bash and Make, respectively (step 1). Then, when the build process starts, *utrace* and *ktrace* traces are collected (step 2). From *utrace*, we could construct the location mapping (step 3). Meanwhile, based on *ktrace*, we could conduct the system call tracing based localization as in RepTrace.

To make the discussion self-contained, we briefly explain how the localization works as in RepTrace. RepTrace relies on the dependency graph, which is constructed by applying differential analysis over the system call traces between the two rounds of build (step 4). For example, the dependency 209174 → 209175 is

[8]http://www.dwarfstd.org

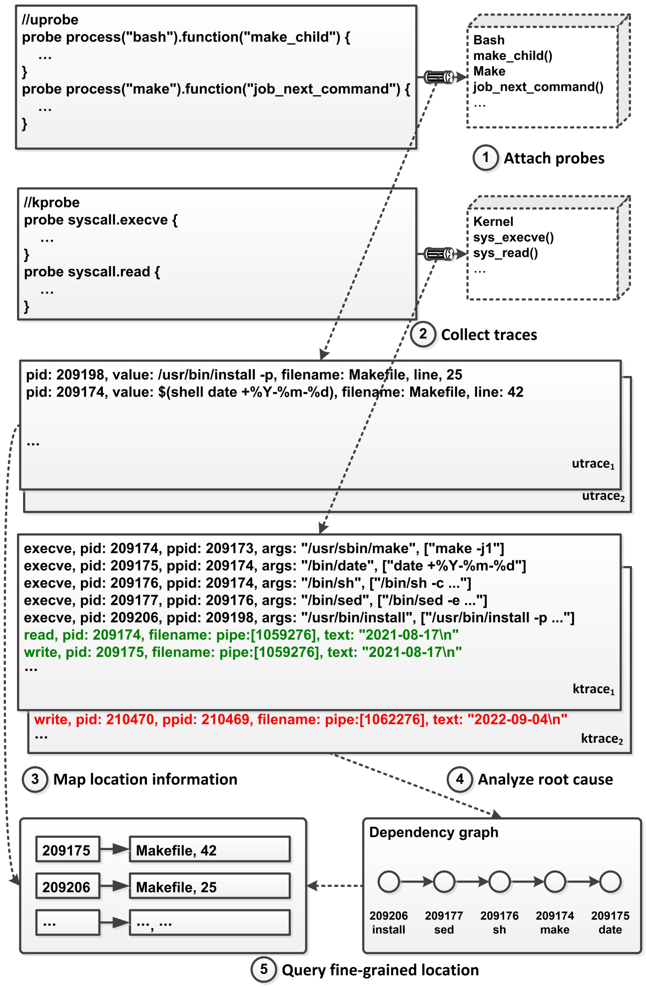


Figure 4: Illustration for the Tracing-based Fine-grained Localization Component

established in that the process with pid 209175 (the *date* command) writes different content (see the last line of *ktrace*) between builds, and the content is written through a pipe (`pipe:[1059276]`) to the process with pid 209174 (the `make` build command). The other dependencies could be detected in a similar way. After traversing all the related traces, we could obtain the dependency graph. From the dependency graph, we could observe that the root cause for the unreproducible build is the *date* command, which is propagated to the inconsistent artifact via the `sed`, `sh`, and `install` commands. Furthermore, with the location mapping, we could decide the problematic *date* command is invoked at line 42 of the `/Makefile` (step 5).

3.2 History-Based Patch Generation

After the fine-grained localization, we proceed to generate the patch to solve the unreproducible build issues. The essential idea of the patch generation process is to utilize the existing patches accumulated by the software repositories. For example, after 8 years

Algorithm 2: PatchGen

Input: Source files *src*, Localization result *res*, Build configuration *conf*, Number of evaluations k
Output: Patch *patch*

1 **begin**
2   *patches* ← **Load-Patches**()
3   *templates* ← $\emptyset$
4   **for** *each* $p \in$ *patches* **do**
5   t ← **Initialize-Template**(p)
6   *templates* ← *templates* $\cup \{t\}$
7   **end**
8   **for** *location* $\in$ *res* **do**
9   *cmd* ← **Extract-Command**(*location*, *src*)
10     $t_{max} \leftarrow \arg\max_{t \in templates} (\textbf{Similarity}(t, cmd))$
11   *operations* ← **Edit-Distance**(t_{max})
12   *patched-cmd* ← **Apply-Operation**(*cmd*, *operations*)
13   *patch* ← **Diff**(*cmd*, *patched-cmd*, *src*)
14   *status* ← **Evaluate**(*patch*, *src*, *conf*)
15   **if** *status* = ***reproducible*** **then**
16   **return** *patch*
17   **end**
18     $k \leftarrow k - 1$
19   **if** $k \leq 0$ **then**
20   **return** *empty-patch*
21   **end**
22   **end**
23 **end**

```
-DATETIME := $(shell date +%Y-%m-%d)
+DATETIME := $(shell date -u -d '@${SOURCE_DATE_EPOCH}' +%Y-%m-%d)
```

Figure 5: Snippet of template for *mylvmbackup*

of the reproducible builds practices lead by Debian, there exists thousands of patches for solving the unreproducible build issues[8].

Given an unreproducible package, we intend to retrieve the most relevant patches, and examine the possibility of transplanting the patch to solve the unreproducible build issue. More specifically, the patch generation process is described in Algo. 2. The algorithm takes the source files, the localization result, the build configuration, and the maximum number of evaluations as inputs, and generate patches that are potentially able to solve the unreproducible build issues of the package. First, we load the patches, which are obtained from Debian's bug tracking system (lines 1–3). For each patch, we extract the commands, and instantiate a template (lines 4–7). Each template consists of a command pair, i.e., the source and destination commands, that describe the modification to the source command.

After that, we initiate the patch generation process (lines 8–22). For each location reported by the localization component, we extract the command to be patched from the source files. Then, we could retrieve the most relevant template t_{max}, with respect to the text similarity between the command to patch and the templates' text. In this study, we consider the n-gram based Cosine similarity

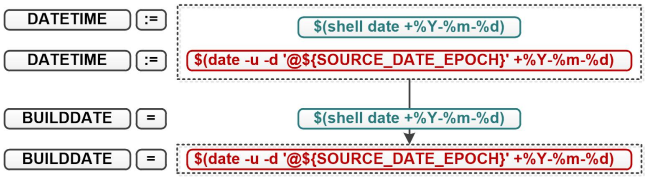


Figure 6: Illustration for command patching

```
--- mylvmbackup-0.15.orig/Makefile
+++ mylvmbackup-0.15/Makefile
@@ -39,7 +39,7 @@

 NAME = mylvmbackup
 VERSION = 0.15
-BUILDDATE = $(shell date +%Y-%m-%d)
+BUILDDATE = $(shell date -u -d '@${SOURCE_DATE_EPOCH}' +%Y-%m-%d)
 MAN1 = man/$(NAME).1
 HOOKS := $(wildcard hooks/*.pm)
 DISTFILES = \
```

Figure 7: Patch for *mylvmbackup*

between command texts[9]. With the retrieved template, we tokenize the extracted commands, calculate the edit operations between the source and the destination commands of t_{max}, and apply the operations to generate the patch[10] (lines 12–13). In this study, we tokenize the extracted commands with respect to the grammar of Make and Bash, based on the following two considerations. On the one hand, we do not directly treat the patches as plain text, in that text based edit operations may not be precise enough. On the other hand, due to the inherent complexity of the grammars for Bash and Make, it is very challenging to parse the patches properly [40].

To validate the generated patch, we evaluate the patch by building the patched source files twice with the configurations, and calculating the checksums of the built artifacts (lines 14–17). If the build is reproducible, the generated patch is returned. Otherwise, the iteration continues with other locations for patching, until the maximum number of evaluations is reached. In this study, RepFix adopts two criteria to determine if the reproducibility of the patched source files: (1) no artifacts are missing (e.g., caused by incorrect build command), and (2) bit-for-bit identical artifacts are generated between the two rounds of build during validation. Besides, we should note that, during the experiments, manual check for the patches are required, in that plausible patches might be generated, e.g., patches with malformed build commands that fail to compile, but with bit-for-bit identical artifacts obtained.

Running Example: Consider the *mylvmbackup* package we introduce in Section 2 again, we first illustrate the relevant patch for the package, as shown in Fig. 5. Then, we tokenize the patch, to obtain the edit operations between the source and the destination command in the patch, i.e., from the `date` command to `SOURCE_DATE_EPOCH` [11]. According to the reproducibility validation tool chain, when building packages, the build time is assigned

[9]https://pypi.org/project/strsimpy. In our preliminary experiments, we observe that RepFix is not sensitive to the choice of similarity.
[10]https://pypi.org/project/python-Levenshtein/

```
-BUILDDATE = $(shell date +%Y-%m-%d)
+BUILDDATE = $(shell date +%u -Y-%m-%d)
```

Figure 8: Snippet of a plausible patch for *mylvmbackup*

to an environment variable `$SOURCE_DATE_EPOCH`, which could be exported when validating the subsequent builds. Hence, replacing the `date` command to `$SOURCE_DATE_EPOCH` could keep the output identical, once the environment variable is set with the same value. Besides, "`-u`" indicates Coordinated Universal Time, which suppresses the influence of timezone. By applying the patch shown in Fig. 6, the *mylvmbackup* package could be reproducibly built. As a comparison, Fig. 8 presents an example of a plausible patch, in which the patched command (`date +%u -Y-%m-%d`) is malformed. Under such circumstance, `date +%u -Y-%m-%d` generates empty output, and identical artifact `/usr/bin/mylvmbackup` is obtained with unexpected content, due to the incorrect patch.

4 EXPERIMENTS

To evaluate the proposed RepFix framework from various perspectives, extensive experiments are conducted. More specifically, we consider the following four research questions (RQs):

- **RQ1**: Is RepFix effective in fixing unreproducible builds for real-world packages?
- **RQ2**: How effectively can the tracing-based fine-grained localization and the history-based patch generation mechanisms improve the overall solution quality?
- **RQ3**: How efficient is each component of RepFix?
- **RQ4**: Is RepFix able to be applied to other unreproducible build packages that have not been previously fixed?

Among these RQs, RQ1 evaluates RepFix's ability to accurately generate valid patches, to resolve the unreproducible build issues. RQ2 concentrates on the contribution of each component of RepFix. By comparing each component with its variant, we could gain more insights into the reason why RepFix works. RQ3 investigates the overhead caused by each component of RepFix. Finally, RQ4 focuses on the generalization of RepFix.

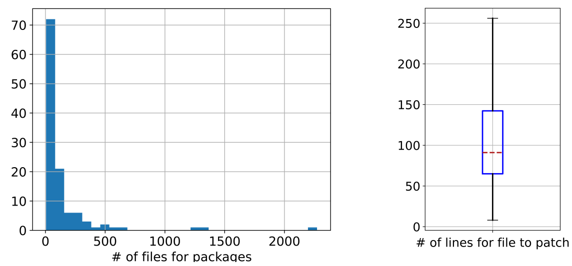
RepFix is implemented in Python 3.9. In particular, the localization component is realized following RepTrace in Java 1.8, in which the tracing tool is switched from strace to SystemTap. For the patch generation component, the build command processing is based on *bashlex*[11], and the parameter k is set with 20. All the experiments are conducted on an Intel NUC (i7-8809G@3.10GHz CPU, 32GB RAM), running Debian (bullseye/amd64).

For the real-world unreproducible packages, we consider the dataset as in RepTrace [38]. There are initially 180 packages in the dataset. However, due to the upgrade of the build tool chain, 7 packages could be reproducibly built, and 57 packages could not be built due to broken dependencies. As a result, the dataset in our experiments contains 116 packages that cannot be reproducibly built. Besides, the patches are obtained from Debian's bug tracking

[11]https://pypi.org/project/bashlex/

system[12]. After filtering out the patches not recognized by *bashlex*, we obtain 1,658 templates[13].

4.1 Investigation of RQ1



(a) Number of files (b) Number of lines

Figure 9: Statistics of the packages in this study

In Fig. 9(a), we illustrate the statistics of the packages. From the figure, we could observe that, the number of files for the packages we fix ranges from less than 100 to over 2,000, with an average number of 139.47. The large number of files poses great challenges for the localization task. Furthermore, we have to identify the line of build command, after the file has been successfully located. In Fig. 9(b), we present the boxplot depicting the distribution of the number of lines for the files to be patched. We could observe that for the majority of the packages, the number of lines ranges within [50, 250]. Moreover, in Fig. 10, we present the proportions of the reasons for the unreproducible build issues. From the figure, we could observe that the majority of the unreproducible packages are caused by timestamp-related issues, e.g., the timestamp in compressed files and the embedded output of the *date* command. This phenomenon conforms with the observation as in the existing literature[28].

Over the 116 packages, RepFix is able to construct valid patches for the 64 packages, i.e., RepFix is able to fix at least one unreproducible issue over these packages. Note that there might be multiple inconsistent artifacts in a single unreproducible package. Hence, we indicate those packages for which part of but not all issues are fixed as partial fixes. In this study, there are 33 packages that are fully reproducible after applying RepFix. Among these 64 fixable or partially fixable packages, 62 of the fixed packages belong to the timestamps category (41 for compressed-file, and 21 for embedded-date). There are also two packages for which the unreproducible build issues are caused by file ordering. The reason is that, in the history-based patch generation, we could not generate valid patches if there are not similar patches with the same root causes. Besides, there are no packages from the randomness category, in that the root causes for these packages mostly lie in Python or Perl scripts, e.g., non-deterministic hash table traversal, which could not be handled by RepFix.

For the successful fixes, we are further interested in the impact of number of templates used by RepFix. As discussed in Section 3,

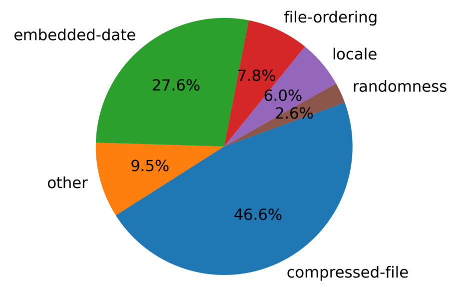


Figure 10: Reasons of unreproducible builds in the dataset

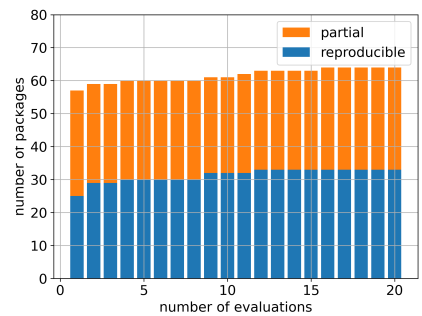


Figure 11: Impact of the maximum evaluation number

during the patch generation procedure, for each localization result, we consider the most relevant existing patches as the templates. In Fig. 11, we present the trend of successful patches as the maximum number of evaluations for each package increases. From the figure, we could observe that, RepFix is not very sensitive to the maximum number of evaluations. Even if when only single template is considered for each package, RepFix is able to fix at least one unreproducible build issue over 57 packages.

Answer to RQ1: In this RQ, we investigate the effectiveness of RepFix, over a set of 116 real-world packages. For 64 of the packages, RepFix is able to at least fix one issue that is responsible for the unreproducible builds. In particular, RepFix successfully makes 33 packages reproducible.

4.2 Investigation of RQ2

To gain more insights into why RepFix works, in this RQ, we investigate each component of RepFix with its variant. More specifically, two comparative approaches are considered.

First, to examine the effectiveness of the tracing-based fine-grained localization, we consider the text-similarity-based localization as the baseline (indicated as Loc(text)).

The comparative line-level localization is intuitively realized as follows. After obtaining the problematic build command and the located file to patch with RepTrace, we extract all the lines from each problematic file, and calculate its cosine-similarity (same metric as in Section 3.2) with the problematic build command. Then, the most similar line is returned as the localization result.

To evaluate the effectiveness of RepFix's localization component, we measure the accuracy rate, precision, recall, and Mean

[12]https://tests.reproducible-builds.org/debian/index_bugs.html
[13]Note that since both the dataset and the patches are from Debian, during the patch generation for each package, we avoid using its corresponding template.

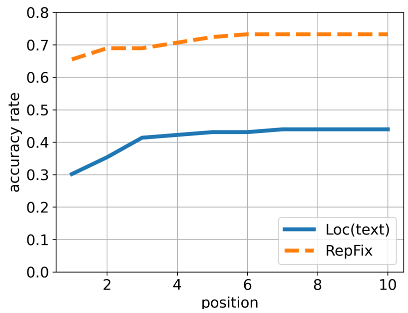


Figure 12: Comparison for the trend of accuracy rate

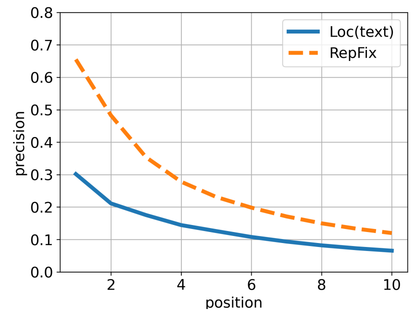


Figure 13: Comparison for the trend of precision

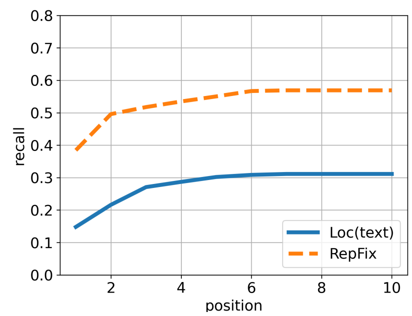


Figure 14: Comparison for the trend of recall

Reciprocal Rank (MRR) in identifying the location for patching for unreproducible builds. The metrics are computed by examining the ranked line-level location for patching returned by the baseline. The Top-N locations in the ranked result list are called the retrieved list, and are compared with the relevance list to compute the accuracy rate, the precision, and the recall, respectively (indicated as $A@N$, $P@N$, and $R@N$, respectively). In particular, $A@N$ measures the percentage of packages for which the Top-N list provides at least one correct location for patching [43]. Finally, MRR is also considered as an aggregate metric to evaluate the retrieved result list[17], which is calculated as:

$$MRR = \frac{1}{|P|}\sum_{i=1}^{|P|}\frac{1}{rank_i}, \tag{1}$$

where $|P|$ indicates the number of packages in the dataset, and $rank_i$ refers to the rank position of the first correct location for patching for the ith package.

In Tab. 1, we present the comparison between the localization component of RepFix and the baseline approach. In the table, we consider the precision, the recall, the accuracy rate for the Top-1, Top-5, and Top-10 results, as well as the MRR metric. From the table, it is obvious that the tracing-based localization outperforms the baseline approach Loc(text) significantly. Over 76 out of the 116 packages, the location reported by RepFix is correct, considering only the Top-1 results. When we further consider the Top-10 results, RepFix successfully locates at least one correct location for patching over 85 packages. To depict the comparison more intuitively, in Figs. 12–14, we illustrate the trends of accuracy rate, precision, and recall, for the comparative localization, which is based on text similarity. From the figures, we could observe that, the results of the baseline localization approach Loc(text) is not satisfying. The recall remains the same for retrieved lists with length larger than 6. Even if we consider the Top-10 result, the recall value remains below 0.4, implying that we could not hit the real location to patch over all the packages with the baseline approach. These phenomena confirm the necessity of applying the tracing-based localization.

Second, to examine the history-based patch generation component, we are interested in whether the token-based command patching is more effective than the baseline in which text-based patching. To achieve this, the baseline adopts the tracing-based fine-grained localization, and replace the patch generation component with text-based modification (indicated as RepFix(text)). More specifically, during patch generation, the patches are treated as text, without taking the grammar of Bash and Makefile into consideration. Given a new unreproducible package, we first sort all the patches according to the tree edit distance between the source command and the command to patch. For the top ranked patch, we generate a sequence of edit operations with the popular Levenshtein edit distance [13, 22], and try applying the operations to the command to patch. Thereafter, we apply the modifications, generate the patch, and validate the patched source files as in RepFix. If the validation succeeds, the patch is returned. Over the dataset, RepFix(text) generates valid patches for 54 packages, which to some extent demonstrates the importance of the token-based patch generation. Similar with RQ1, we also present the impact of the maximum number of evaluations for RepFix(text) in Fig. 15. From the figure, we could observe that RepFix(text) is not as effective as RepFix, especially if the maximum number of evaluates is limited.

Answer to RQ2: In this RQ, we focus on why RepFix works. By comparing the localization of RepFix with Loc(text), we confirm the effectiveness of the fine-grained localization. Also, RepFix is able to fix unreproducible issues over 10 more packages than RepFix(text), which demonstrates the usefulness of the token-based patch generation.

Table 1: Results of RepFix and Loc(text) for the line-level localization task

Approach	*A@1*	*A@5*	*A@10*	*P@1*	*P@5*	*P@10*	*R@1*	*R@5*	*R@10*	*MRR*
RepFix	0.6552	0.7241	0.7328	0.6552	0.2310	0.1198	0.3844	0.5504	0.5691	0.6816
Loc(text)	0.3017	0.4310	0.4397	0.3017	0.1259	0.0655	0.1480	0.3020	0.3114	0.3528

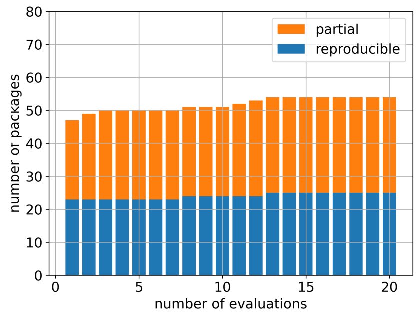


Figure 15: Impact of the maximum evaluation number for RepFix(text)

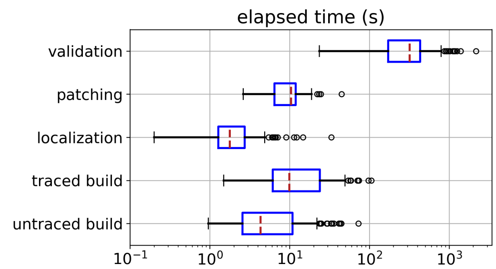


Figure 16: Time consumed by components of RepFix

4.3 Investigation of RQ3

In this RQ, we investigate the efficiency of RepFix. As mentioned in Section 3, RepFix leverages SystemTap to realize the runtime trace collection, which introduces non-negligible time cost during the build process. Hence, we shall empirically investigate the impact of the tracing on the build performance. Also, we are interested in the time elapsed for each main steps of RepFix.

Fig. 16 illustrates the distribution of each component's time elapsed over the dataset. In the figure, each boxplot corresponds to one or more step in Fig. 3, i.e., the boxplots indicate the traced build (step 1), the localization (step 2), the patching (steps 3–6, in that these steps are closely related), and the validation (step 7). Besides, we also present the distribution of the build time without tracing, to analyze the impact of system call tracing. From the figure, we observe that the validation is the most time-consuming step, with an average value of 364.40 seconds. This observation is as expected, since in the iterative fixing paradigm, validation has to be conducted for each generated patch. Meanwhile, the other steps tend not to be very time-consuming. Interestingly, when we compare the build time with/without tracing, we observe that the average time under the two circumstances is 18.47 and 9.17 seconds, respectively. Such phenomenon implies that the tracing time cost is not negligible, but is acceptable in most cases.

Answer to RQ3: By comparing the time distribution of each main step of RepFix, we identify that the patch validation is the most time-consuming step. Also, we confirm that the overhead of the tracing for both the kernel and the user-space applications is non-negligible. However, with the promising fine-grained localization ability, we think the time cost is in general acceptable.

4.4 Investigation of RQ4

Finally, in RQ4, we are interested in applying RepFix over new unreproducible packages. As a case study, we test RepFix over real-world Arch Linux packages. The reason for the choice of the repository is that the Arch Linux community is actively conducting the reproducible builds validation and fixing practice. Also, Arch Linux is a rapidly evolving GNU/Linux distribution, where we could receive efficient feedback from the developers and maintainers.

The package fixing procedure is carried out as follows. First, the package status is obtained from the status page of Arch Linux's continuous integration testing system[14]. Then, we download the source packages for the packages that are unreproducible with Arch Linux's build source management tool *asp*[15]. After that, we filter out the packages that use build systems other than Bash and Make, since currently RepFix does not handle build systems like Cargo and Bazel.

In total, there are four packages for which RepFix is able to resolve the unreproducible issues. We submitted the patches in the form of bug reports to Arch Linux's bug tracking system [3–6]. All the bug reports have been assigned, and two of the patches have been accepted.

```
--- when-1.1.40.orig/Makefile
+++ when-1.1.40/Makefile
@@ -49,7 +49,7 @@ install: when.1
    install -m 755 temp $(DESTDIR)$(bindir)/when
    # ... 755=u:rwx,go:rx
    rm temp
-   gzip -9 <when.1 >when.1.gz
+   gzip -9n <when.1 >when.1.gz
    - test -d $(DESTDIR)$(MANDIR) || mkdir -p $(DESTDIR)$(MANDIR)
    install -m 644 when.1.gz $(DESTDIR)$(MANDIR)
    rm -f when.1.gz
```

Figure 17: Patch for *when*

In particular, within the packages for which RepFix successfully generated patches, the package *when*[16] was an interesting case. The package (with version 1.1.40-2) was unreproducible due to the misuse of *gzip* argument, i.e., *gzip* by default keeps its timestamp in

[14]https://tests.reproducible-builds.org/archlinux/state_FTBR.html.
[15]https://github.com/archlinux/asp
[16]https://archlinux.org/packages/community/any/when/

the compressed file, unless the -n argument is used [7]. After validating the patch shown in Fig. 17 locally, we submitted the patch with a bug report [3]. However, the patch was not immediately accepted. Instead, the maintainer suggested reporting the patch upstream. The reason, as stated by the documentation[17], was that fixing the issue upstream might help other downstream repositories as well. Following the suggestion, we opened an issue at the package's repository at GitHub[18]. Also, we explained to the author why the patch should be applied. Finally, the patch was accepted, which was then pushed to Arch Linux's repository later, and the bug report was closed.

Answer to RQ4: RepFix is able to effectively solve unreproducible build issues for real-world packages. Four patches are submitted to the Arch Linux repository, and two patches have been accepted.

5 DISCUSSION

5.1 Extensibility of RepFix

RepFix could be potentially extended from two aspects.

First, in this study, we demonstrate the flexibility of RepFix with two types of build scripts, i.e., Bash and Make. To support more types of build systems, the user-space tracing is necessary. In most build systems, it is common that build tools are responsible for maintaining the relationship between build command, location information, as well as build processes when invoked. Hence, it would be feasible to design SystemTap probes to capture such data structures, to construct the location mapping for RepFix. Once such connection is established, the corresponding build system could be supported. Furthermore, for other operating systems, such as BSD distributions and Windows, there also exist system-level monitoring facilities such as DTrace [2] and Event Tracing for Windows (ETW) [1], with which fine-grained localization could be realized.

Second, in this study, we rely on the existing patches to construct new patches for new unreproducible build issues. A more effective way might be summarizing a set of template-based formal rules, to guide the generation of patches. In such paradigm, we might be able to improve the generalization of RepFix over new repositories, since with summarized rules, we are able to take more domain-specific knowledge into consideration.

5.2 Threats to Validity

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

First, during the patch generation component, after a candidate patch is generated, the validation step is conducted, to evaluate whether the patch is valid. It is possible that the patched source files could be reproducibly built, yet the functionality is not the same as in the original version[32]. For example, if the patched command fails to be compiled, and generates nothing, the built package might be reproducible, but the patch is not acceptable. To mitigate this threat, we introduce a constraint in patch validation, checking the existence of all the built artifacts. Moreover, we manually check the patches that pass the validation.

Second, another threat arises within the dynamic tracing framework. In this study, we employ SystemTap to collect traces from both kernel and user-space applications. At heavy workload, SystemTap may skip certain probes, so that the traces might be incomplete. To avoid such circumference, we make sure that no multiple builds are executed simultaneously. Also, sufficient buffer is assigned to SystemTap. In our experiment, no missing probes are discovered. A possible approach to preventing this issue is to implement the dynamic tracing tool from scratch based on the ptrace system call [10] as in strace [12] and DetTrace [36].

6 RELATED WORK

There are two topics that are closely related to this study, i.e., the work related to reproducible builds, and the work related to build script analysis and repair.

6.1 Reproducible Builds

Software reproducibility is an emerging research topic, that has attracted great interests. Lamb and Zacchiroli[28] from the Debian community make a systemic review of the current state of the reproducible builds. As of the fixing of unreproducible builds, currently the existing studies focus on the localization task. In 2018, Ren et al. [37] propose the initial work RepLoc that focuses on the automated localization for unreproducible builds. In their study, the localization for unreproducible builds is modeled as an information retrieval task, and a hybrid framework that combines heuristic filtering and query expansion is developed, in search of the problematic files that cause the build to be unreproducible. In 2019, a system-call-tracing-based approach RepTrace is proposed [38], which features the ability of root cause analysis for unreproducible builds. With the dependency graph constructed based on the system call traces, deeper insights could be gained into why builds are unreproducible.

Besides, there also exist studies that intend to guarantee the software build process to be reproducible. Navarro Leija et al. [36] propose the framework DetTrace, a reproducible container abstraction for Linux implemented in user space. With DetTrace, the reproducibility of software build could be ensured by intercepting all the system calls that may introduce non-determinism. Similarly, He et al. [24] develop ConstBin, which tries to fix unreproducible issues during the build process, by capturing and replacing arguments of the execve system calls for suspicious build commands.

For these studies related to reproducible builds, RepLoc and RepTrace concentrate on the localization task, but are not able to realize the fixing functionality. Meanwhile, DetTrace and ConstBin intend to fix unreproducible issues on-the-fly during the build process. Despite the promising achievements, the software reproducibility property could not be realized without the tools, i.e., the build process has to be conducted under the supervision of DetTrace or ConstBin. Unlike these approaches, RepFix is able to generate patches for the packages. Once fixed, no more containers or tools are required for future builds. Also, the generated patches in upstream repositories could benefit their downstream repositories.

6.2 Build Script Analysis and Repair

Due to the inherent complexity of build systems, many software packages suffer from build failures, and great effort has to be made to fix build scripts. In recent years, there have been a series of

[17] https://wiki.archlinux.org/index.php/Bug_reporting_guidelines
[18] https://github.com/bcrowell/when/issues/22

studies on the analysis and the repairing of the build scripts. On the one hand, to effectively analyze build scripts, various techniques have been applied. For example, SyMake [42] apply static analysis such as symbolic evaluation to help developers better understand build scripts. Gazzillo [20] proposes KMAX, to find all configurations of Linux kernel's kbuild Makefiles. Besides, there are also dynamic analysis approaches such as MKCHECK [30] and BUILDFS [41].

Compared with these studies on build script analysis, a unique feature of this study lies in its ability to utilize the runtime trace capturing from both the kernel and the user-space applications, i.e., Make and Bash. With the modern tracing framework, more accurate runtime behavior could be captured, with which we are able to realize fine-grained localization.

On the other hand, to fix build script faults, there have been growing research interests on the automated repairing of build scripts. Foyzul and Wang [23] propose the HIREBUILD framework, which is an automatic approach to history-driven repair of build scripts. Lou et al. [33] develop HOBUFF, which considers the historical projects, as well as the present project under test and external resources. In 2020, Lou et al. [34] systematically investigate more than 1,000 build issues from Stack Overflow, to summarize fix patterns for different types of failure, with respect to three well-known build systems, i.e., Maven, Ant, and Gradle.

These studies focus on the fixing of build failures, i.e., HIREBUILD and HOBUFF are applied when projects failed to build from source. In contrast, REPFIX is more targeted to the scenario of unreproducible builds.

7 CONCLUSIONS

In this paper, we propose the initial work REPFIX to generate patches for unreproducible builds in an automated paradigm. The framework features the combination of the tracing-based fine-grained localization and the history-based patch generation. On the one hand, with the unified tracing tool SystemTap, the system call trace induced dependency graph could be associated with the user-space trace guided line-level localization, and tackle the localization granularity challenge. On the other hand, by utilizing the existing patches, we are able to generate valid patches for real-world unreproducible packages. Furthermore, REPFIX successfully fix the unreproducible build issues of four Arch Linux packages that have not been previously fixed. The patches are submitted to Arch Linux's bug tracking system, and two patches have been accepted.

For future work, we are interested in the possibility of automatically generating fixing rules, from the existing patches that solve unreproducible builds. Also, an empirical study to gain deeper insights into the fixed patches is also an interesting direction. Besides, we would like to extend the fixing technique to more software repositories which have not considered reproducible builds practice.

ACKNOWLEDGEMENTS

Zhilei Ren is also affiliated with Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology. This work is supported in part by the National Natural Science Foundation of China under Grants 62132020, 62072068, 62032004, 61872273, 62141221, and Fundamental Research Funds for the Central Universities (NO. NJ2020022).

REFERENCES

[1] 2021. About Event Tracing. https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing. Accessed: 2021-09-02.
[2] 2021. DTrace. http://dtrace.org. Accessed: 2021-09-01.
[3] 2021. FS#69535: when. https://bugs.archlinux.org/task/69535. Accessed: 2021-09-02.
[4] 2021. FS#70302: pythia8. https://bugs.archlinux.org/task/70302. Accessed: 2021-09-02.
[5] 2021. FS#70303: zssh. https://bugs.archlinux.org/task/70303. Accessed: 2021-09-02.
[6] 2021. FS#71953: dd_rescue. https://bugs.archlinux.org/task/71953. Accessed: 2021-09-02.
[7] 2021. GNU gzip: General file (de)compression. https://www.gnu.org/software/gzip/manual/gzip.html. Accessed: 2021-04-3.
[8] 2021. History of reproducible builds. https://reproducible-builds.org/docs/history/. Accessed: 2021-08-30.
[9] 2021. mylvmbackup. https://tracker.debian.org/pkg/mylvmbackup. Accessed: 2022-02-01.
[10] 2021. ptrace(2) Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html. Accessed: 2021-08-21.
[11] 2021. The SOURCE_DATE_EPOCH specification. https://reproducible-builds.org/docs/source-date-epoch/. Accessed: 2021-09-02.
[12] 2021. Strace. https://strace.io. Accessed: 2021-04-17.
[13] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Massimiliano Di Penta. 2013. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *2013 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 230–239.
[14] Raymond Chen. 2018. Why are the module timestamps in Windows 10 so nonsensical? https://devblogs.microsoft.com/oldnewthing/20180103-00/?p=97705. Accessed: 2021-08-31.
[15] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. 2019. Detecting memory errors at runtime with source-level instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 341–351.
[16] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. 2018. Run-time detection of protocol bugs in storage I/O device drivers. *IEEE Transactions on Reliability* 67, 3 (2018), 847–869.
[17] Nick Craswell. 2009. Mean Reciprocal Rank. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer US, Boston, MA, 1703–1703.
[18] Jake Edge. 2017. Reproducible builds. https://lwn.net/Articles/719823/. Accessed: 2021-08-27.
[19] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. 2018. On early detection of application-level resource exhaustion and starvation. *Journal of Systems and Software* 137 (2018), 430–447.
[20] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, 279–290.
[21] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. Unix shell programming: the next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 104–111.
[22] Jiatao Gu, Changhan Wang, and Junbo Zhao. 2019. Levenshtein Transformer. *Advances in Neural Information Processing Systems* 32 (2019), 11181–11191.
[23] Foyzul Hassan and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1078–1089.
[24] Hongjun He, Jicheng Cao, Lesheng Du, Hao Li, Shilong Wang, and Shengyu Cheng. 2020. ConstBin: A Tool for Automatic Fixing of Unreproducible Builds. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 97–102.
[25] Jingzhu He, Ting Dai, and Xiaohui Gu. 2018. Tscope: Automatic timeout bug identification for server systems. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 1–10.
[26] Ryan Hurst. 2021. Verifiable design in modern systems. https://security.googleblog.com/2021/07/verifiable-design-in-modern-systems.html. Accessed: 2021-08-31.
[27] Pascal Jungblut, Roger Kowalewski, and Karl Fürlinger. 2018. Source-to-Source Instrumentation for Profiling Runtime Behavior of C++ Containers. In *2018 IEEE 20th International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 948–953.
[28] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* (2021). https://doi.org/10.1109/MS.2021.3073045 Early Access.
[29] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of*

the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 169–180.

[30] Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1234–1244.

[31] Chang Liu, Zhengong Cai, Bingshen Wang, Zhimin Tang, and Jiaxu Liu. 2020. A protocol-independent container network observability analysis system based on eBPF. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 697–702.

[32] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 615–627.

[33] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 43–54.

[34] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding Build Issue Resolution in Practice: Symptoms and Fix Patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Virtual Event, USA). ACM, 617–628.

[35] Douglas H Martin, James R Cordy, Bram Adams, and Giulio Antoniol. 2015. Make it simple-an empirical analysis of gnu make feature use in open source projects. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 207–217.

[36] Omar S Navarro Leija, Kelly Shiptoski, Ryan G Scott, Baojun Wang, Nicholas Renner, Ryan R Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 167–182.

[37] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 71–81.

[38] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. 2019. Root cause localization for unreproducible builds via causality analysis over system call tracing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 527–538.

[39] Young Shi, Mingzhi Wen, Filipe Roseiro Cogo, Boyuan Chen, and Zhen Ming Jack Jiang. 2021. An Experience Report on Producing Verifiable Builds for Large-Scale Commercial Systems. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3092692

[40] Paul D. Smith. 2004. Makefile grammar. https://www.mail-archive.com/help-make@gnu.org/msg02778.html. Accessed: 2021-08-23.

[41] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A Model for Detecting Faults in Build Specifications. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 144 (Nov. 2020), 30 pages.

[42] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. 2012. Build code analysis with symbolic evaluation. In *34th International Conference on Software Engineering (ICSE)*. IEEE, 650–660.

[43] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. 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, 689–699.

[44] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2021), 332–347.