Automated Localization 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

He Jiang
Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, School of Software, Dalian University of Technology, Dalian, China
jianghe@dlut.edu.cn

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

Zijiang Yang
Department of Computer Science, Western Michigan University, Kalamazoo, MI, USA
zijiang.yang@wmich.edu

ABSTRACT

Reproducibility is the ability of recreating identical binaries under pre-defined build environments. Due to the need of quality assurance and the benefit of better detecting attacks against build environments, the practice of reproducible builds has gained popularity in many open-source software repositories such as Debian and Bitcoin. However, identifying the unreproducible issues remains a labour intensive and time consuming challenge, because of the lacking of information to guide the search and the diversity of the causes that may lead to the unreproducible binaries.

In this paper we propose an automated framework called RepLoc to localize the problematic files for unreproducible builds. RepLoc features a query augmentation component that utilizes the information extracted from the build logs, and a heuristic rule-based filtering component that narrows the search scope. By integrating the two components with a weighted file ranking module, RepLoc is able to automatically produce a ranked list of files that are helpful in locating the problematic files for the unreproducible builds. We have implemented a prototype and conducted extensive experiments over 671 real-world unreproducible Debian packages in four different categories. By considering the topmost ranked file only, RepLoc achieves an accuracy rate of 47.09%. If we expand our examination to the top ten ranked files in the list produced by RepLoc, the accuracy rate becomes 79.28%. Considering that there are hundreds of source code, scripts, Makefiles, etc., in a package, RepLoc significantly reduces the scope of localizing problematic files. Moreover, with the help of RepLoc, we successfully identified and fixed six new unreproducible packages from Debian and Guix.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; Software evolution;

KEYWORDS

Unreproducible Build; Localization; Software Maintenance

ACM Reference Format:
Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated Localization for Unreproducible Builds. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180224

1 INTRODUCTION

As an indicator of the ability that the binaries could be recreated consistently from source, recent years have witnessed the emerging idea of reproducible builds. Given the source files, the reproducibility is described as the ability of building identical binary under pre-defined build environments [15]. In this study, source files include source code, scripts, Makefiles, build configurations, etc [6]. Checking the reproducibility of software creates a verifiable linkage that bridges the gap between the readable source files and the binary packages, which is important from various perspectives.

Firstly, reproducibility is very important for the safety of build environments. For software ecosystems, attacks against the build environment may lead to serious consequences. By compromising the system to produce packages with backdoors [26, 45], malicious behaviors such as trusting trust attack [41] may be introduced during the build time. For example, in 2015, over 4,000 iOS applications were infected by a counterfeit version of Apple's Xcode development environment (known as XcodeGhost) [1]. XcodeGhost injected malicious code during compiling time so that developers unknowingly distributed malware embedded in their applications [21]. Obviously, a solution is to ensure that the same source files always lead to the same binary packages so that an infected different binary immediately raises alarms. Unfortunately, a major obstacle of detecting such attacks lies in the transparency gap between the source files and their compiled binary packages. Due to non-deterministic issues such as timestamps and locales, it is not uncommon that rebuilding an application yields different binaries even within secure build environments. Therefore, these kinds of attacks often elude detection because different binaries of the same application is normal.

Besides detecting attacks against build environments, validating the reproducibility is also helpful in debugging and finding certain release-critical bugs (e.g., `libical-dev` 1.0-1.1) [2]. Furthermore, in the context of and continuous integration and software upgrade [37], reproducible packages could be helpful in caching, and reducing redundant operations, e.g., by eliminating the necessity of delivering the different binaries compiled from the same source files. Due to the significant benefits, many open-source software repositories have initiated their validation processes. These repositories include GNU/Linux distributions such as Debian and Guix, as well as software systems like Bitcoin [19]. For instance, since 2014, the number of Debian's reproducible packages has been steadily increasing. Figure 1 presents the trend of the reproducible builds in Debian [14]. As of August 2017, over 85% of Debian's packages could be reproducibly built.

Despite the effort towards reproducibility, many packages remain unreproducible. For example, according to Debian's Bug Tracking System (BTS), as of August 23, 2017, there are 2,342 packages that are not reproducible [14] for the unstable branch targeting the AMD64 architecture. Such large number of unreproducible packages implies the challenges in detecting and then fixing the unreproducible issues. In particular, the localization task for the problematic files is the activity of identifying the source files that cause unreproducibility, which ranks source files based on their likelihood of containing unreproducible issues. Currently, the localization task is mostly manually conducted by developers. Since there may be hundreds to thousands of source files for a package, the localization tends to be labor intensive and time consuming.

To address this problem, we consider the source files as text corpus, and leverages the diff log[1] generated by comparing the different binaries to guide the search. As such, the localization of the problematic files can be modeled as a classic Information Retrieval (IR) problem: given the source files and the diff log, determine those problematic files from the source files that are relevant to the unreproducible issues. The IR model has the potential to automate the localization task. However, the localization task is challenging, due to its unique characteristics.

First, the information for locating the problematic files within the source files is very limited. The diff log generated by comparing the different binaries, which is considered as the input of the IR process, may not be sufficiently informative. We call this challenge an **information barrier**. In addition, there are many causes that may lead to unreproducible builds, such as embedding timestamps in files and recording file lists in non-deterministic order. The detailed issues are manually listed in Debian's documentation [12]. Moreover, the diverse types of files in a package also add to the complexity of localizing the problematic files, which may reside in not only the source code, but also other types of files such as scripts, Makefiles and build configurations. We call this challenge a **diverse-cause barrier**.

To break through the barriers, we propose a localization framework called RepLoc that targets the localization task in search of problematic files for unreproducible builds. Given an unreproducible package with two different built binaries as the input, RepLoc produces a list of ranked source files. RepLoc features two components that address the two aforementioned challenges. For the information barrier, we develop a Query Augmentation (QA)

[1]Generated by `diffoscope`, https://diffoscope.org

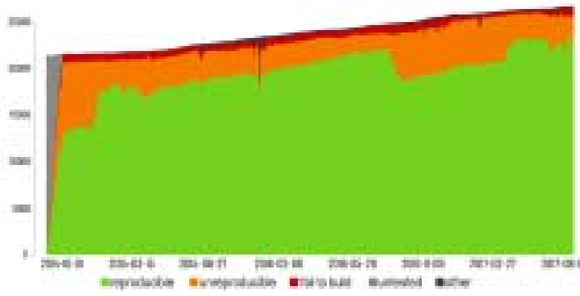


Figure 1: Reproducibility status of Debian unstable for AMD64

component that utilizes the information extracted from the build logs to enhance the quality of the queries (represented by the file names extracted from the diff logs, see Section 2). For the diverse-cause barrier, we develop a Heuristic rule-based Filtering (HF) component. More specifically, we propose 14 heuristic rules that are obtained by summarizing the information presented in Debian's documents. Furthermore, we employ a weighted File Ranking (FR) component to combine the QA and HF components, and build an integrated framework to automate the localization of the problematic files for unreproducible builds.

To evaluate RepLoc, we have collected a real-world dataset that consists of 671 unreproducible packages. Since these packages were later fixed with patches from Debian's BTS, we know exactly which files caused the unreproducibility and thus can use the facts to evaluate the accuracy of RepLoc. If we consider the topmost ranked file only, RepLoc achieves an accuracy rate of 47.09%. If we expand the range to include top ten ranked files, the accuracy rate becomes 79.28%. For other metrics such as precision and recall, RepLoc also outperforms the comparative approaches significantly. To further evaluate the effectiveness of our approach, we use RepLoc on unreproducible packages that have never been fixed before. With the help of RepLoc, we successfully identified the problematic files, then manually fixed the unreproducible issues over three Debian packages. Moreover, the usefulness of RepLoc is examined over a different software repository (Guix [11] in this study). Under the guidance of RepLoc, problematic files for three unreproducible packages from Guix are detected and fixed.

This paper makes the following main contributions.

- To the best of our knowledge, this is the first work to address the localization task for unreproducible builds.
- We propose an effective framework RepLoc that integrates heuristic filtering and query augmentation. A prototype has been implemented based on the approach.
- We have evaluated RepLoc on 671 unproducibile packages that were later fixed in the Debian repository. The experimental results show that RepLoc is effective. We have made the benchmarks publicly available at **https://reploc.bitbucket.io**.
- Under the guidance of RepLoc, we fixed six unreproducible packages from Debian and Guix, and submitted the patches to the BTSs of the two repositories. Among the submitted patches, four have been accepted.

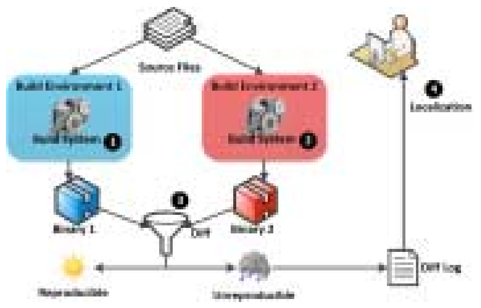


Figure 2: Reproducibility validation work flow

Table 1: Snippet of altered environment variations

Configuration	First build	Second build
env TZ	"/usr/share/zoneinfo/Etc/GMT+12"	"/usr/share/zoneinfo/Etc/GMT-14"
env LANG	"C"	"fr_CH.UTF-8"
env LANGUAGE	"en_US:en"	"fr_CH:fr"
env BUILDDIR	"/build/1st"	"/build/2nd"
…	…	…

The rest of this paper is organized as follows. In Section 2, we give the background of this work. Our approach is presented in Section 3, followed by experimental study in Section 4. The threats to validity and related work are described in Sections 5–6. Finally, Section 7 concludes the paper.

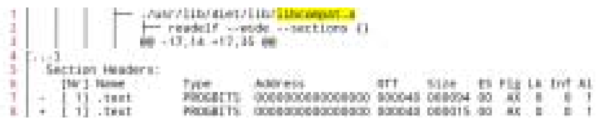
2 BACKGROUND

Taking Debian as a typical example, Figure 2 illustrates the common work flow of validating the reproducibility of packages [17]. First, the source files are compiled under two pre-defined build environments (steps 1–2). More specifically, the build environments are constructed by setting up altered environment variables or software configurations. For instance, within Debian's continuous integration system,[2] altered environment variables include locales, timezones, user privileges, etc. Table 1 presents a snippet of the altered environment (see [22] for more detailed information). Two versions of binaries can be generated with respect to each environment. The two versions are then compared against each other (step 3). If they are not bit-to-bit identical, the localization of problematic files that lead to unreproducible builds is required, based on the diff log and the source files (step 4).

The build and the comparison procedures (steps 1–3) can easily be automated, but the localization (step 4) mainly relies on the developers. Unfortunately, manual effort to identify the files that lead to unreproducible builds is nontrivial. As shown in Figure 2, the diff logs are the major source of the information to guide the localization of the problematic files, which, unfortunately, are not always sufficiently informative.

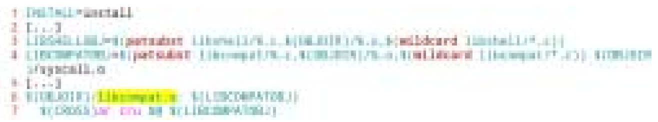
Figure 3 gives a snippet of the diff log for `dietlibc`, a libc implementation optimized for small size. In the original version (0.33~cvs20120325-6), a static library file differs between the two versions during the build time (`/usr/lib/diet/lib/libcompat.a`). As shown in the diff log, `diffoscope` indicates the difference via the output of the GNU binary utility `readelf`. However, since the

[2]https://jenkins.debian.net



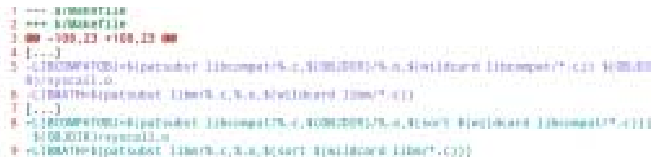
```
1 │ │ │ ├── /usr/lib/diet/lib/libcompat.a
2 │ │ │ │ ├── readelf --wide --sections {}
3 │ │ │ │ @@ -17,14 +17,15 @@
4 [...]
5   Section Headers:
6     [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
7 -   [ 1] .text             PROGBITS        0000000000000000 000040 000034 00  AX  0   0  1
8 +   [ 1] .text             PROGBITS        0000000000000000 000040 000015 00  AX  0   0  1
```

Figure 3: Diff log snippet for `dietlibc`



```
1 INSTALL=install
2 [...]
3 LIBSHELLOBJ=$(patsubst libshell/%.c,$(OBJDIR)/%.o,$(wildcard libshell/*.c))
4 LIBCOMPATOBJ=$(patsubst libcompat/%.c,$(OBJDIR)/%.o,$(wildcard libcompat/*.c)) $(OBJDIR
  )/syscall.o
5 [...]
6 $(OBJDIR)/libcompat.a: $(LIBCOMPATOBJ)
7   $(CROSS)ar cru $@ $(LIBCOMPATOBJ)
```

(a) Makefile snippet



```
1 --- a/Makefile
2 +++ b/Makefile
3 @@ -108,23 +108,23 @@
4 [...]
5 -LIBCOMPATOBJ=$(patsubst libcompat/%.c,$(OBJDIR)/%.o,$(wildcard libcompat/*.c)) $(OBJDI
  R)/syscall.o
6 -LIBMATH=$(patsubst libm/%.c,%.o,$(wildcard libm/*.c))
7 [...]
8 +LIBCOMPATOBJ=$(patsubst libcompat/%.c,$(OBJDIR)/%.o,$(sort $(wildcard libcompat/*.c)))
   $(OBJDIR)/syscall.o
9 +LIBMATH=$(patsubst libm/%.c,%.o,$(sort $(wildcard libm/*.c)))
```

(b) Patch snippet

Figure 4: Makefile and patch snippet for `dietlibc`

diff content may not be well comprehensible (e.g., lines 7–8 in Figure 3), we do not leverage such information in this study. Meanwhile, Figure 4 presents a snippet of a problematic file (`/Makefile`) and the patch that fixes the issue. In Figure 4(b), line 8 indicates that the root cause of the unreproducibility lies in the non-stable order of the object files, which are fed to the `ar` utility to generate `libcompat.a` (lines 6–7 of Figure 4(a)). The difficulty in this example is that, the diff log may fail to provide sufficient information. Though it is possible to match the correct file with only the file name, i.e., line 6 of Figure 4(a), chances are that other irrelevant files containing the same file name might be matched as well.

The aforementioned example illustrates how problematic files can be detected and fixed. In reality there are multiple altered build configurations and can be many corresponding causes that lead to unreproducible builds. For example, changing the timezone environment variable (`env TZ`) may cause the C/C++ packages that embed `__DATE__` macro to be unreproducible, and the locale environment variable (`env LC_*`) may trigger unreproducible issues of packages that capture the text generated by programs. These diverse unreproducible causes make the localization task difficult.

3 OUR APPROACH

In this section, we discuss the details of RepLoc. Figure 5 depicts the work flow of RepLoc that consists of three components QA, HF, and FR. For each component, we shall explain its design and implementation, companioned with the intermediate results over the running example `dietlibc`.

3.1 Query Augmentation Component

The upper part of Figure 5 depicts the QA component, which enriches the queried information by matching the files in the diff log and the build logs, to tackle the information barrier.

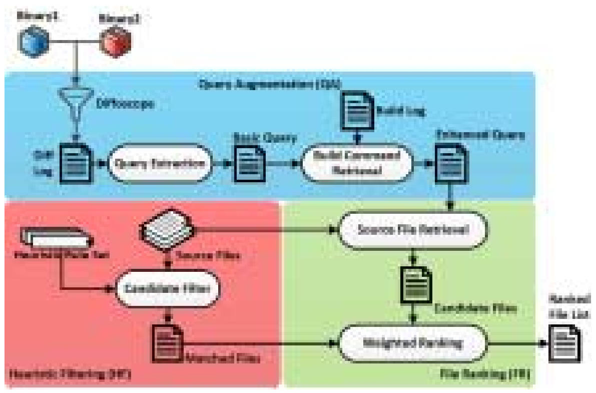
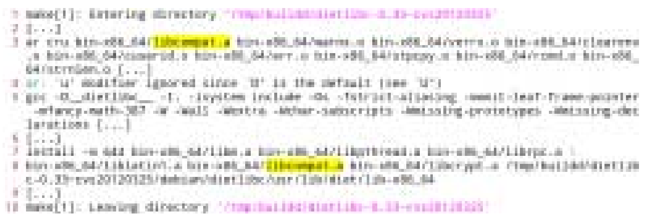


Figure 5: The RepLoc Framework



```
1  make[1]: Entering directory '/tmp/buildd/dietlibc-0.33~cvs20120325'
2  [...]
3  ar cru bin-x86_64/libcompat.a bin-x86_64/warns.o bin-x86_64/verrs.o bin-x86_64/clearenv
   .o bin-x86_64/cuserid.o bin-x86_64/err.o bin-x86_64/stpcpy.o bin-x86_64/rcmd.o bin-x86_
   64/strnGen.o [...]
4  ar: 'u' modifier ignored since 'D' is the default (see 'U')
5  gcc -D__dietlibc__ -I. -isystem include -Os -fstrict-aliasing -momit-leaf-frame-pointer
    -mfancy-math-387 -W -Wall -Wextra -Wchar-subscripts -Wmissing-prototypes -Wmissing-dec
   larations [...]
6  [...]
7  install -m 644 bin-x86_64/libc.a bin-x86_64/libpthread.a bin-x86_64/librpc.a \
8  bin-x86_64/liblatin1.a bin-x86_64/libcompat.a bin-x86_64/libcrypt.a /tmp/buildd/dietlib
   c-0.33~cvs20120325/debian/dietlibc/usr/lib/diet/lib-x86_64
9  [...]
10 make[1]: Leaving directory '/tmp/buildd/dietlibc-0.33~cvs20120325'
```

Figure 6: Build log snippet for `dietlibc`

First, the diff log is generated using `diffoscope`. Then, the query extraction module takes the diff log as the input, and generates the basic query. In this study, the basic query consists of the file names in the diff log. As mentioned, due to the information barrier, the information that can be utilized to localize the problematic files is limited other than a list of files that are different within the two build processes. Thus, we enhance the quality of the queries with the build command retrieval module. The motivation for this module is that, during the build process, the build information such as the executed commands can be obtained. Moreover, based on the co-occurrence relationship between the file names in the diff log and the build commands, we can identify the build commands with which the files mentioned in the diff log are built. Hence, it is rational to augment the query by supplementing the build commands from the build log.

Figure 6 illustrates a snippet of the build log of the exemplifying package `dietlibc`. It can be observed that the build log is more informative and provides supplementary information with respect to the diff log. More specifically, we first split the build log into build command segments, with respect to the "`Entering/Leaving directory`" tags generated by `make` (e.g., lines 1 and 10 of Figure 6). With this operation, the commands invoked under the same directory can be grouped together, as a file of the augmentation corpus (denoted as a command file). Note that though there are two versions of build logs with respect to the two build environments, since we are interested in the build command, the choice of either version of build log does not have an impact on the results. Then, the relevant files in the corpus are obtained by utilizing an IR model. In essence, any IR model can be adopted. In this study, we employ the Vector Space Model (VSM), due to its simplicity and effectiveness.

Table 2: Heuristic rule set

ID	Rule	PCRE statement
1	`TIME_MACRO`	**__TIME__**
2	`DATE_MACRO`	**__DATE__**
3	`GZIP_ARG`	\b**gzip**\s(?!.*-[a-z9]*n)
4	`DATE_CMD`	(\$\(**date**)|(\$\(shell\s***date**)|(\`**date**)
5	`PY_DATE`	**datetime**\.**datetime**\.**today**
6	`PL_LOCALTIME`	\$\.***localtime**
7	`SYSTEM_DATE`	**system**.***date**
8	`DATE_IN_TEX`	\\date.*\\today
9	`SORT_IN_PIPE`	^.*\\|'(?!.*LC_ALL=).*\s***sort**\b
10	`GMTIME`	**gmtime**\(
11	`TAR_GZIP_PIPE`	\b**tar**\b.*\\|\s*\b**gzip**\b
12	`PL_UNSORTED_KEY`	(^(?!.***sort**).*\s***keys**\s*%)
13	`LS_WITHOUT_LOCALE`	^.*\$\(.*(?!.*LC_ALL=).*\s*\b**ls**\b
14	`UNSORTED_WILDCARD`	(^(?!.***sort**).*\s*\b**wildcard**\b)

To realize the VSM based augmentation, we calculate the cosine similarity between basic query and the command files. Thereafter, the matched commands from the most relevant command files are obtained. In particular, for the VSM model, we assign weight value for each file with the *TF-IDF* (Term Frequency-Inverse Document Frequency) measurement, which is widely used in IR [32]. In this paper, for a term t in a document d, its $TF\text{-}IDF_{t,d}$ value is calculated based on $f_{t,d} \times \frac{N}{n_t}$, where $f_{t,d}$ indicates the number of t's occurrences in d, n_t denotes the number of files in which t appears, and N means the number of source files. With *TF-IDF* defined, each file is represented as a vector, and the cosine similarity with the basic query is used to rank the command files.

$$Sim(\vec{l}, \vec{s}) = \frac{\vec{l} \cdot \vec{s}}{|\vec{l}||\vec{s}|}, \tag{1}$$

where $\vec{l} \cdot \vec{s}$ represents the inner product of the basic query and the command file, and $|\vec{l}||\vec{s}|$ denotes the product of 2-norm of the vectors. After that, the basic query and the retrieved contents, which are commands executed during the build process, are concatenated together as the enhanced query.

Running example: For `dietlibc`, all the file names in the diff log, e.g., `./usr/lib/diet/lib/libcompat.a`, are extracted as the basic query. Then, within the augmentation, `ar cru bin-x86_-64/libcompat.a [. . . ]` (line 3 of Figure 6) and the build commands in the same command file are retrieved. Finally, the contents of the retrieved command files are appended after the basic query, as the final query.

3.2 Heuristic Filtering Component

The HF component is designed to capture the problematic files by incorporating the domain knowledge, which is represented as frequently observed patterns. In HF, the heuristic rules are constructed based on the following criteria: (1) The rules are manually constructed based on Debian's documentation [13]. (2) The rules are summarized for the four major categories of unreproducible issues (see Setcion 4.2). We traverse the notes in the documentation, and capture those issues that are described as Perl Compatible Regular Expression (PCRE). For example, invoking `gzip` without "-n" argument could be expressed using the negative assertions feature of PCRE (rule 3 in Table 2). Meanwhile, as a counterexample, the timestamps embedded in Portable Executable (PE) binaries are hard to be identified by heuristic rules or even by developers [20]. After manual inspection based on the criteria, we obtain 14 heuristic rules, which are presented in Table 2, and described as follows:

(1) `TIME_MACRO`: using C time preprocessing macro in source files will embed different timestamps when compiled at different times. (2) `DATE_MACRO`: embedding C date preprocessing macro in source files is similar as the previous case. (3) `GZIP_ARG`: if applying gzip without -n argument, timestamps will be embedded in the header of the final compressed file. (4) `DATE_CMD`: capturing the current date with the **date** shell command. (5) `PY_DATE`: obtaining date time in Python scripts. (6) `PL_LOCALTIME`: obtaining date time in Perl scripts. (7) `SYSTEM_DATE`: recording system time in the compiled binary. (8) `DATE_IN_TEX`: embedding date in TeX files, which influences the built pdf files. (9) `SORT_IN_PIPE`: execute **sort** in pipeline without locale setting. (10) `GMTIME`: obtaining current date time. (11) `TAR_GZIP_PIPE`: execute **tar** and **gzip** in pipeline. (12) `PL_UNSORTED_KEY`: traversing unsorted hash keys in Perl script does not guarantee identical order. (13) `LS_WITHOUT_LOCALE` : capturing **ls** without locale setting is similar with `SORT_IN_PIPE`. (14) `UNSORTED_WILDCARD`: using **wildcard** in Makefiles without sorting, similar with `PL_UNSORTED_KEY`.

By applying the rules over the source files (e.g., with GNU `grep -r -P`), we obtain a subset of files that may lead to unreproducible builds. Note that these rules equally treat the source files as plain text, rather than consider the file types (e.g., based on file extension). The reason is that the unreproducible issues may reside in snippets or templates that do not follow file extension conventions, which are eventually embedded into unreproducible binaries. Based on such consideration, we do not sort the matched files in HF.

Running example: For `dietlibc`, there are in total five problematic files, namely, `/libpthread/Makefile`, `/libdl/Makefile`, `/debian/{rules, implicit}`, and `/Makefile`. Among these files, `/Makefile` (see Figure 4(b)) can be captured by the `UNSORTED_WILDCARD` rule, in which `sort` does not appear before `wildcard`. However, we should note that there may be false alarms, e.g., for unexecuted commands or text in the comments. Consequently, HF may fail to place the matched problematic files at the top of the list.

3.3 File Ranking Component

The motivations behind the combination of HF and QA are twofold: (1) The heuristic rules in HF focus on the static aspect of the source files, i.e., treat all the source files in a unified way, and capture the suspicious files that match the defined patterns. Such mechanism can handle various file types. Unfortunately, there may be false alarms, especially for those files unused during the build process. (2) The build log based augmentation takes the dynamic aspect of the build process into consideration. With QA, we concentrate on the commands invoked during the build process. Hence, by combining the mechanisms, we can strengthen the visibility of the problematic files that lead to unreproducible builds.

In the FR component, these goals are realized as follows. First, with the augmented query, the relevant files are obtained with the source file retrieval module. Similar as in Section 3.1, the VSM model is adopted to calculate the similarity values between the augmented query and each source file. Second, since we have acquired both the files retrieved by HF and the similarity values between source files

Algorithm 1: RepLoc

```
   Input: binary package first, binary package second, weight $\alpha$
   Output: candidate file list result
1  begin
       // Query Augmentation
2      log ← diffoscope(first, second)
3      query ← parse_log(log)
4      command_files ← parse_build_log(build_log)
5      relevant_command ← retrieve_relevant(query, command_files)
6      augmented ← concatenate(query, relevant_commant)
       // Heuristic Filtering
7      $list \leftarrow \emptyset$
8      for each source file $s$ do
9          if $s$ is matched by any rule in Table 2 then $list \leftarrow list \cup \{s\}$
10     end
       // File Ranking
11     for each source file $s$ do
12         if $s \in list$ then $w_s \leftarrow 1$
13         else $w_s \leftarrow 0$
14         $score_s$ ← Calculate $Sim'$ with respect to Equation 2
15     end
16     return sort(source_files, score)
17 end
```

Table 3: Files retrieved by RepLoc and its components over `dietlibc`, with successful hits in bold

Rank	FR (without QA)	Rank	FR (with QA)
1	/CHANGES	1	**/debian/rules**
2	**/debian/rules**	2	**/Makefile**
3	**/Makefile**	3	/CHANGES
4	/debian/control	4	/debian/patches/0005-[…].diff
5	/FAQ	5	/diet.c
Rank	**HF**	**Rank**	**RepLoc**
1	/t.c	1	**/debian/rules**
2	**/debian/implicit**	2	**/Makefile**
3	/debian/dietlibc-dev.postinst.in	3	/CHANGES
4	**/debian/rules**	4	**/libpthread/Makefile**
5	/libugly/gmtime.c	5	**/libdl/Makefile**

and the augmented query, in the file ranking module, it is natural to combine these two types of information, to better capture the problematic files. For example, we can modify Equation 1 and apply Sim' to rank the source files:

$$Sim'(\vec{l}, \vec{s}) = (1 - \alpha) \times Sim(\vec{l}, \vec{s}) + \alpha \times w_s, \qquad (2)$$

where $w_s = 1$ for those source files matched by the HF component, and $w_s = 0$ otherwise. $\alpha \in [0, 1]$ is a weight parameter to balance the two terms, e.g., large α values make RepLoc favor the HF component.

With Equation 2, the source files are ranked according to their modified similarity to the augmented query, and the top ranked files are returned as the final results of RepLoc. We should note that, in this study, we adopt the file-level localization paradigm, in that the fixing for many unreproducible packages is not unique. For instance, statements declaring missing environmental variables can appear anywhere in the file before it is needed. Hence, it is difficult to establish line-level ground-truth. In Algorithm 1, we present the pseudo-code of RepLoc, which combines QA (lines 2–6), HF (lines 7–10), and FR (lines 11-16) sequentially.

Running example: In Table 3, we present the top five files retrieved by RepLoc and its individual components. From the table, we can observe that without augmenting the query, FR is able to retrieve two problematic files. However, the topmost ranked file is a changelog (`/CHANGES`), in that the file names in the diff log appear in this file. In contrast, with the query augmented, FR (with QA) is able to rank the two problematic files at the top of the list. Meanwhile, although HF is able to capture `/libpthread/Makefile`, the file is not assigned top rank due to other false alarms, e.g., `/t.c`. Finally, by combining FR, QA, and HF, RepLoc is able to locate four problematic files.

4 EXPERIMENTAL RESULTS

4.1 Research Questions

In this study, we intend to systematically analyze RepLoc, by investigating the following Research Questions (RQs):

- RQ1: Is RepLoc sensitive to the weighting parameter α?
- RQ2: How effective is RepLoc?
- RQ3: How efficient is RepLoc?
- RQ4: Is RepLoc helpful in localizing unfixed packages?

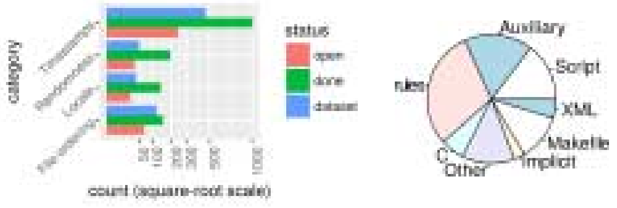
Among these RQs, RQ1 concentrates on the impact of the weighting scheme between the components in RepLoc. RQ2 focuses on how well RepLoc performs in terms of different quality metrics. RQ3 examines whether RepLoc is time consuming, and RQ4 investigates the RepLoc's generalization.

4.2 Data Preparation

In this study, the dataset is constructed by mining Debian's BTS. To the best of our knowledge, Debian is the only repository providing both past-version packages and reproducibility-related patches, which are crucial for generating the corpus and the ground truth. Consequently, all the packages within the dataset are extracted from Debian' BTS, which are tagged as unreproducible by bug reporter via `debtags`, i.e., the command line interface for accessing the BTS. According to Debian's documentation, there are 14 categories of reproducible issues [16]. There are also two special categories indicating the packages that fail to build from source, and the toolchain issues (non-deterministic issues introduced by other packages, see Section 5), which are not considered in this study.

We download all the 14 categories of 1716 bug reports, and download the packages, with their corresponding patches. Then, we apply the validation tool kit,[3] to obtain the corresponding diff logs and build logs. In this study, we consider those categories with more than 30 packages. With such criterion, we obtain 671 packages in the dataset, which fall into the four largest categories. Figure 7(a) illustrates the statistics of the dataset. In the figure, we present the numbers of the open and closed bugs in Debian's BTS, as well as the number of packages in the dataset. Among the four categories of packages, the Timestamps category contains the most packages (462), followed by File-ordering (118), Randomness (50), and Locale (41). For all the four categories of 1491 packages that are labeled as "done", the packages in the dataset take a portion of 45.34%. Note that there are less packages in the dataset than closed bug reports, since packages may not be compilable due to the upgrade of their dependencies.

[3]The tool kit realizes steps 1–3 of Figure 2, available at https://anonscm.debian.org/cgit/reproducible/misc.git



(a) Number of packages (b) Problematic file types

Figure 7: File statistics of the dataset

In Figure 7(b), we illustrate the statistics of the patches in the dataset. From the figure, we could observe that there are many types of files that might be involved in the unreproducible builds. For these files, the Debian `rules` files, which are the main build scripts, take the largest portion of the fixed files (29.82%). Auxiliary files, such as the `configure` scripts and input files (`*.in`), takes the second largest portion (17.21%). After that, there are the Makefiles (11.68%), scripts such as Python/Perl/PHP files (14.60%), C/C++ files (5.94%), XML files (4.80%), implicit build files (2.71%). Since we classify the files based on their file extensions heuristically, there are also 13.24% of the files that are not easy to classify, e.g, those without file extensions. This phenomenon conforms with the second barrier mentioned in Section 1, i.e., the causes to the unreproducible builds are diverse, which makes the localization task very challenging.

4.3 Implementation and Metrics

RepLoc is implemented in Perl 5.24 and Java 1.8. All the experiments are conducted on an Intel Core i7 4.20 GHz CPU server with 16 GB memory, running GNU/Linux with kernel 4.9.0. For the comparative algorithms, we consider four variants of RepLoc, since there is no prior approach addressing this problem. The first two variants implement two baseline algorithms, which only consider either the HF or the FR model (denoted as RepLoc(HF) and RepLoc(FR)). These two variants are incorporated to examine the performance of its building-block components. Moreover, RepLoc(FR) could be considered the simulation of the manual localization, since in FR, the retrieval is realized by matching source files with diff log contents. Then, RepLoc(FR+QA) considers utilizing the QA component to enhance the basic queries extracted from the diff logs. Finally, RepLoc indicates the version discussed in Section 3.

To evaluate the effectiveness of RepLoc, metrics commonly used in the IR literatures are employed to evaluate the performance of RepLoc, including the accuracy rate, the precision, the recall, and the Mean Average Precision (*MAP*). The metrics are computed by examining the ranked list of source files returned by the framework in response to a query. The Top-*N* source files in the ranked list is called the retrieved set and is compared with the relevance list to compute the Precision and Recall metrics (denoted by *P@N* and *R@N* respectively). Given an unreproducible package with problematic files, a Top-*N* accuracy rate score, e.g. *A@1*, *A@5*, and *A@10*, of a localization tool is the portion of Top-*N* lists a tool provides that at least one problematic file contains in it [30, 48]. In this study,

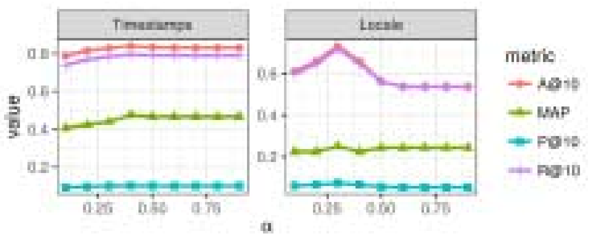


Figure 8: Impact of varying α

we also report *P@1*, *P@5*, *P@10* and *R@1*, *R@5*, *R@10* [28, 48]. *P@N* means the portion of problematic files successfully retrieved in a Top-N list, while *R@N* measures how many problematic files are retrieved in a Top-N list among all the problematic files:

$$P@N = \frac{\text{\# of files that cause unreproducible builds}}{N}, \quad (3)$$

$$R@N = \frac{\text{\# retrieved problematic files in the Top-}N\text{ list}}{\text{\# of problematic files}}. \quad (4)$$

Precision and Recall usually share an inverse relationship, in that, the Precision is higher than Recall for lower values of N and vice versa for higher values of N. An overall metric of retrieval accuracy is known as Mean Average Precision (*MAP*), which is the average of the Average Precision (*AP*) values over all the problematic files in unreproducible packages. For an unreproducible package with several problematic files, the *AP* is computed as $\sum_{k=1}^{M} \frac{P@k \times pos(k)}{\text{\# of files related in the patch}}$, where M is the size of a ranking list, $pos(k)$ indicates whether the kth file in a ranking list is related to the unreproducible build, and $P@k$ is the precision described in Equation 3. With *AP* defined, *MAP* can be calculated by averaging all the *AP* scores across all the unreproducible packages.

4.4 Investigation of RQ1

In this RQ, we intend to investigate whether RepLoc is sensitive to the weighting parameter α. As described in Section 3, in Equation 2, we propose the weighted similarity between queries and source files. Hence, in this RQ, we are interested in investigating RepLoc's behavior as we alter the weight of the two components. More specifically, for each category of dataset, we randomly select half of the packages, and a grid search from 0.1 to 0.9 with a step of 0.1 is employed to analyze the impact of varying α.

Considering the Timestamps and the Locale datasets, we visually present the trend of the *A@10*, *P@10*, *R@10* and the *MAP* values against the α value in Figure 8. From the figure, the following observations can be drawn. First, for the randomly selected packages, the performance of RepLoc exhibits similar trend, i.e., when α is set within the range $[0.2, 0.4]$, RepLoc obtains the best results. Second, we observe that RepLoc is not very sensitive to α, unless α is too large, which will make RepLoc prefer the HF component. Hence, for the subsequent experiments, α is set with 0.3.

Answer to RQ1: Experimental results show that, RepLoc is not very sensitive to the parameter, which to some extent demonstrates the robustness of RepLoc.

4.5 Investigation of RQ2

In this RQ, we examine whether RepLoc locates the problematic files accurately. We present the experimental results, and discuss the phenomena observed. In Table 4, we first give the results over the datasets. The table is organized as follows. The first column indicates the four categories of datasets we built in this study (see Section 4.2). The second column represents the four variants of RepLoc. Then, the rest of the table presents the metrics that evaluate the performance of each variant. Note that for the accuracy rate, the precision, and the recall, the metric values are averaged over all the packages. Besides, we also present the aggregate performance at the bottom of the table.

Taking the Timestamps dataset as an example, several interesting phenomena can be observed. First, the performance of RepLoc(HF) is not satisfying. Even considering the Top-10 results, the corresponding accuracy rate is around 70%. To examine the representativeness of the heuristic rules, in Table 5 we present the results of RepLoc(HF) with single rule. We report the *A@10*, *P@10*, *R@10*, and *MAP* of the five rules that perform the best. Among the rules, the `GZIP_ARG` rule achieves the highest accuracy rate. However, the *A@10* value is below 30%, which is significantly outperformed by RepLoc(HF) that considers all the rules. Similar observations could be drawn for other performance metrics, which to some extent confirms the diverse-cause barrier.

Second, by comparing the results of RepLoc(FR+QA) against RepLoc(FR) in Table 4, we can confirm the usefulness of QA. As mentioned, RepLoc(FR) could be loosely considered the simulation of manual localization, which tries to match the problematic files with the diff log contents. Over the Timestamps dataset, *A@10* of RepLoc(FR) is 71.21%. With the augmentation of the query, *A@10* improves to 76.41%. Moreover, when we combine RepLoc(FR+QA) with HF, the performance is further improved, i.e., *A@10* of RepLoc achieves 82.90%, which implies that for over 80% of the unreproducible packages in the Timestamps dataset, at least one problematic file is located in the Top-10 list. Besides, similar results are obtained over the other datasets, i.e., RepLoc(HF) and RepLoc(FR) perform the worst, RepLoc(FR+QA) outperforms RepLoc(FR) considering the *A@10* value, and RepLoc performs the best.

Associated with Table 4, we also conduct statistical tests, to draw confident conclusions whether one algorithm outperforms the other. For the statistical test, we employ the Wilcoxon's signed rank test, with a null hypothesis stating that there exists no difference between the results of the algorithms in comparison. We consider the 95% confidence level (i.e., p-values below 0.05 are considered statistically significant), and adopt the *P@10* and *R@10* as the performance metrics. We do not consider the accuracy rate and the MAP metrics, in that these are aggregate metrics. Over all the instances, when comparing RepLoc with any of the other three baseline variants, the null hypothesis is rejected (p-value < 0.05 for both *P@10* and *R@10*), which implies that RepLoc outperforms their baseline variants in a statistically significant way.

To gain more insights into the behavior of RepLoc, we present the performance of the four variants against the number of retrieved results in Figure 9, over typical datasets. In the figure, the x-axis and the y-axis indicate the number of retrieved files, and the performance metrics. From the sub-figures, we confirm that over both

Table 4: Comparison results between RepLoc and its variants

Dataset	Model	*A@1*	*A@5*	*A@10*	*P@1*	*P@5*	*P@10*	*R@1*	*R@5*	*R@10*	*MAP*
Timestamps	RepLoc(HF)	0.4048	0.6775	0.7229	0.4048	0.1511	0.0835	0.3587	0.6222	0.6682	0.3522
	RepLoc(FR)	0.3160	0.5736	0.7121	0.3160	0.1268	0.0807	0.2821	0.5253	0.6553	0.2777
	RepLoc(FR+QA)	0.4762	0.6753	0.7641	0.4762	0.1511	0.0883	0.4155	0.6177	0.7102	0.4009
	RepLoc	**0.5238**	**0.7792**	**0.8290**	**0.5238**	**0.1792**	**0.0991**	**0.4538**	**0.7295**	**0.7839**	**0.4400**
File-ordering	RepLoc(HF)	0.3136	0.4407	0.4576	0.3136	0.0983	0.0534	0.2653	0.3968	0.4197	0.2528
	RepLoc(FR)	0.1525	0.5169	0.6949	0.1525	0.1085	0.0729	0.1215	0.4427	0.6150	0.1136
	RepLoc(FR+QA)	0.3814	0.6780	0.7627	0.3814	0.1492	0.0864	0.3040	0.5978	0.6856	0.2804
	RepLoc	**0.4492**	**0.7288**	**0.7966**	**0.4492**	**0.1661**	**0.0966**	**0.3774**	**0.6506**	**0.7331**	**0.3572**
Randomness	RepLoc(HF)	0.1000	0.2200	0.2600	0.1000	0.0480	0.0280	0.0850	0.2100	0.2500	0.0813
	RepLoc(FR)	0.1000	0.3000	0.4800	0.1000	0.0640	0.0500	0.1000	0.3000	0.4650	0.1000
	RepLoc(FR+QA)	**0.2200**	0.3200	0.4200	**0.2200**	0.0680	0.0460	**0.2100**	0.3050	0.4100	**0.2050**
	RepLoc	0.2000	**0.4200**	**0.5000**	0.2000	**0.0880**	**0.0540**	0.1900	**0.4050**	**0.4900**	0.1854
Locale	RepLoc(HF)	0.0976	0.3171	0.3659	0.0976	0.0634	0.0366	0.0976	0.3049	0.3415	0.0976
	RepLoc(FR)	0.1463	0.2439	0.4634	0.1463	0.0488	0.0463	0.1463	0.2317	0.4512	0.1494
	RepLoc(FR+QA)	0.2439	0.4146	0.5610	0.2439	0.0829	0.0561	0.2317	0.4024	0.5488	0.2256
	RepLoc	**0.2683**	**0.5122**	**0.7317**	**0.2683**	**0.1024**	**0.0732**	**0.2561**	**0.5000**	**0.7195**	**0.2500**
Overall	RepLoc(HF)	0.3472	0.5797	0.6200	0.3472	0.1288	0.0712	0.3059	0.5324	0.5734	0.2990
	RepLoc(FR)	0.2608	0.5231	0.6766	0.2608	0.1142	0.0750	0.2320	0.4760	0.6216	0.2278
	RepLoc(FR+QA)	0.4262	0.6334	0.7258	0.4262	0.1404	0.0829	0.3694	0.5777	0.6736	0.3544
	RepLoc	**0.4709**	**0.7273**	**0.7928**	**0.4709**	**0.1654**	**0.0937**	**0.4087**	**0.6774**	**0.7491**	**0.3949**

Table 5: Result of RepLoc(HF), with single heuristic rule

ID	Rule	*A@10*	*P@10*	*R@10*	*MAP*
3	`GZIP_ARG`	0.2981	0.0341	0.2823	0.1864
4	`DATE_CMD`	0.2191	0.0253	0.1878	0.1250
14	`UNSORTED_WILDCARD`	0.1058	0.0112	0.0968	0.0578
13	`LS_WITHOUT_LOCALE`	0.0671	0.0072	0.0428	0.0247
9	`SORT_IN_PIPE`	0.0387	0.0039	0.0351	0.0261

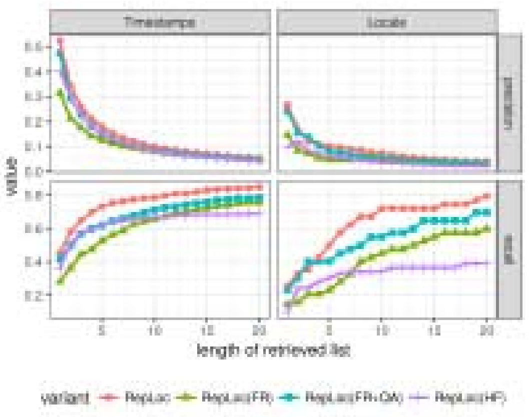


Figure 9: Trends of precision and recall of RepLoc

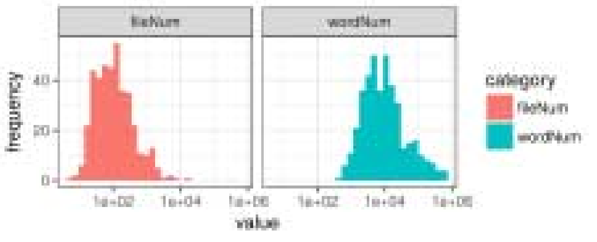


Figure 10: Histogram for scale statistics

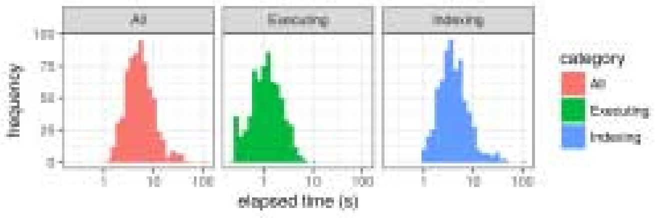


Figure 11: Histogram for efficiency evaluation

the datasets, RepLoc outperforms the other variants significantly, i.e., the performance curves for RepLoc lie above those for other variants, which implies that for all the cases of the retrieved results, combining the two components is able to obtain better results. This phenomenon conforms with our observations in Table 4.

Answer to RQ2: By comparing the variants of RepLoc over 671 real world packages, we confirm that by combining the heuristic rule-based filter and the query augmentation, RepLoc is able to outperform its variants.

4.6 Investigation of RQ3

In this RQ, we evaluate RepLoc from the efficiency perspectives. Since manually localizing the unreproducible issues is a time consuming task, automating such process is profitable only if the proposed approach is time efficient. Hence, we present the time statistics of the experiments. Figure 10 depicts the statistics of the source files as histograms, in which the x-axis indicates the number of source files (fileNum) and the words (wordNum), and the y-axis represents the associated frequency. In this study, the number of files ranges within [6, 19890], and the number of words for the majority of the packages ranges around 1×10^4, which implies that manually inspecting the files would be difficult.

Since the scale of the packages in this study varies greatly, it is intuitive that the localization process over different packages will

Table 6: Results of RepLoc, over `manpages-tr`

Rank	RepLoc(FR)	Rank	RepLoc(FR+QA)
1	/debian/rules	1	/debian/patches/bashisms.patch
2	/source/man8/mount.8.xml	2	/debian/rules
3	/source/tr/linkata.sh	3	**/source/manderle.sh**
4	/source/man1/rsync.1.xml	4	/Makefile
5	**/source/manderle.sh**	5	/debian/manpages-tr.prune

Rank	RepLoc(HF)	Rank	RepLoc
1	/source/man1/gzip.1.xml	1	**/source/manderle.sh**
2	**/source/manderle.sh**	2	/debian/patches/bashisms.patch
3		3	/debian/rules
4		4	/source/man1/gzip.1.xml
5		5	/source/man1/patch.1.xml

vary accordingly. To investigate this issue, we present the results related to time efficiency considering the three variants of RepLoc. In Figure 11, we illustrate the distributions of the dataset scalability and the execution time. In the sub-figures, the x-axis indicates the time in seconds, and the y-axis represents the frequency. From the results, we observe that, the indexing of the documents consumes the largest portion of time, compared with other components. In particular, the median of the execution time for RepLoc is 5.14 seconds.

Answer to RQ3: In this RQ, we investigate the efficiency perspectives of RepLoc. In this study, the indexing of the document consume the majority of the time.

4.7 Investigation of RQ4

For RQ1–RQ3, to evaluate the performance of RepLoc properly, we employ the packages that have been fixed, and adopt the patches from the BTS as the ground truth. However, in the real-world reproducible validation scenario, the patches are not available in advance. Hence, in this RQ, we intend to investigate RepLoc under such condition. More specifically, we consider two scenarios, i.e., we apply RepLoc to the packages over (1) Debian packages that are previously unfixed, and (2) the unreproducible packages from Guix.

First, we are interested in whether RepLoc could be generalized to unfixed packages, which are obtained from the continuous integration system of Debian. We also check the BTS, to ensure that the packages have not been fixed. We apply RepLoc to localize the problematic files, and then manually check and fix the unreproducible issues. Through localization and fixing, 3 unreproducible packages belonging to the Timestamps category are fixed, i.e., `regina-rexx` (3.6-2), `fonts-uralic` (0.0.20040829-5), and `manpages-tr` (1.0.5.1-2). We submit the corresponding patches to the BTS [3–5], and the one for `fonts-uralic` has been accepted.

For these packages, the problematic files are ranked among the top of the retrieved list by RepLoc. For example, in Table 6, we present the results over the package `manpages-tr`. The table is organized similarly as Table. 3. From the table, we observe that RepLoc is able to localize problematic files effectively, i.e., the problematic files are ranked the first in the result. The package is unreproducible due to the invocation of `gzip` without "`-n`", and the issue can be captured by the `GZIP_ARG` rule in "`/source/manderle.sh`". However, since the heuristic rules fail to capture the dynamic aspect of the build process, a file ("`/source/man1/gzip.1.xml`") unused during compilation is also retrieved. In contrast, with FR and QA, we concentrate on the files involved by the build process. By combining

Table 7: Results of RepLoc, over `skalibs`

Rank	RepLoc(FR)	Rank	RepLoc(FR+QA)
1	/package/info	1	/configure
2	/doc/[...]/kolbak.html	2	/src/[...]/uint32_reverse.c
3	/doc/[...]/unixmessage.html	3	/src/[...]/badrandom_here.c
4	/doc/[...]/unix-transactional.html	4	/src/[...]/goodrandom_here.c
5	/doc/[...]/unix-timed.html	5	/src/[...]/md5_transform.c
...	...	...	...
24	**/Makefile**	10	**/Makefile**

Rank	RepLoc(HF)	Rank	RepLoc
1	/tools/gen-deps.sh	1	**/Makefile**
2	**/Makefile**	2	/configure
3	/src/[...]/localtm_from_ltm64.c	3	/src/[...]/uint32_reverse.c
4		4	/src/[...]/badrandom_here.c
5		5	/src/[...]/goodrandom_here.c

both the static (HF) and the dynamic (HF and QA) perspectives, the problematic file is ranked the first of the list with higher probability.

Second, we consider the packages from the Guix repository, to investigate whether the knowledge obtained from Debian could be generalized to other repositories. The reasons we choose Guix are that, (1) the repository is interested in the reproducible builds practice [23], and (2) its package manager provides the functionality of validating package reproducibility locally, which facilitates the experimental design. As a demonstration, we localize and manually fix the problematic files of 3 packages, namely `libjpeg-turbo` (1.5.2), `djvulibre` (3.5.27), and `skalibs` (2.3.10.0). Similar with the previous case, the patches were submitted to Guix's BTS [8–10]. Taking `skalibs` as an example, we present the results of the variants of RepLoc in Table 7. From the table, we could observe that the problematic file "`/Makefile`" is assigned the top rank. Contrarily, without RepLoc, over 900 source files have to be manually traversed. Such observation to some extent demonstrates the usefulness of RepLoc in leveraging the knowledge from Debian to a different repository such as Guix. After localizing the problematic file and manually fixing, the submitted patch has been accepted and pushed into the code base of Guix [10]. Similarly, the patches for `djvulibre` [8] and `libjpeg-turbo` [9] have also been accepted.

Answer to RQ4: We demonstrate that RepLoc is helpful in localizing unfixed unreproducible packages from both Debian and Guix. In particular, unreproducible issues of 6 packages from both repositories are fixed under the guidance of RepLoc, which have not been fixed before this study.

5 THREATS TO VALIDITY

There are several objections a critical reader might raise to the evaluation presented in this study, among which the following two threats deserve special attention.

First, in this study, the heuristic rules in HF are summarized from Debian's documentation. Also, we leverage the build log gathered from the build process. Hence, some may argue that the approach cannot be generalized to other software repositories because it relies too much on Debian's infrastructure. To mitigate this threat, in RepLoc, attention is paid so that the components are not specialized for Debian. For example, despite knowing that the Debian `rules` files take the largest portion of the problematic files (see Figure 7(b)), no extra priority is given to these files during ranking. Also, in HF, we avoid using heuristic rules specific to Debian, and intend to make the rules as general as possible. For instance, UNSORTED_WILDCARD is applicable for Makefile based build systems, and GZIP_ARG is helpful if gzip-based compression is involved. As a result, the results of this study can be generalized to other repositories. As demonstrated in RQ4, we have successfully applied RepLoc to Guix. For other repositories, applying RepLoc should only require minor adaptation. For example, for the Fedora project, the build log can be gathered by parsing the verbose output of the mock build tool, and the diff log could be generated by diffoscope as well.

Second, when constructing the datasets, the unreproducible packages caused by the tool-chain issues are not considered. For these packages, the unreproducible issues are introduced by the depended packages rather than the current package. Hence, identification of the tool-chain issues is another challenging task that requires further manual investigation [7]. Besides, we should note that fixing the tool-chain issues may help make more packages reproducible. For example, when reproducible-related patches were accepted by gcc from upstream, around 200 unreproducible packages that depended on gcc became reproducible automatically [18]. We plan to explore the tool-chain issues in the future.

6 RELATED WORK

6.1 Bug Localization Related Work

First, this study is closely related to the fault localization studies, especially the IR-based approaches.

For example, Zhou et al. [49] proposed a specialized VSM based approach, and consider the similarities between bug reports to localize buggy files. Wang et al. [44] propose a compositional model that integrates multiple variants of VSM. In particular, they model the composition of different VSM variants as a optimization problem, and apply a genetic algorithm to search for the suitable composition pattern between VSM variants. Wang et al. [42] investigate the usefulness of IR-based fault localization techniques, and discover that the quality of the bug reports are crucial to the performance of localization tasks.

Meanwhile, domain knowledge is utilized to improve the performance of IR-based bug localization techniques. Ye et al. [48] find bug-fixing frequency and bug-fixing recency of source code files are helpful for bug localization. Saha et al. [38] find the structure of bug reports and source code files are also good knowledge for bug localization. They consider bug reports or source code files as documents with structured fields, e.g., summary and description, or file name, class name, and method name, respectively. Stack-trace information in bug report is also analyzed [33, 46] to improve the performance of bug localization. Besides, version histories [39, 40, 43] and similar bug reports [24] are proved to be useful.

Besides, with the development of IR techniques, other text mining methodologies are also incorporated to support locating buggy files. For example, due to its effectiveness, Latent Dirichlet Allocation (LDA) has gained its popularity in the field of bug localization. Lukins et al. [31] propose a static LDA-based technique for automatic bug localization. Lam et al. [29] propose a localization framework HyLoc that combines deep learning and IR-based model. They integrate deep neural network and a VSM variant, to complement the two standalone components. Experimental results over real world projects demonstrate that their proposed model outperforms the individual models. Rao et al. [35] propose an incremental framework to update the model parameters of the Latent Semantic Analysis, which is then applied to localize buggy files. Experiments over software libraries with ten years of version history validate their framework.

However, despite the closeness to these studies, we should note that the problem in this study has its unique features. For example, the counterpart of the bug reports in IR-based fault localization, i.e., the diff logs, are not sufficiently informative to guide the retrieval.

6.2 Reproducible Build Related Work

To the best of our knowledge, there have not been studies on localizing files that cause unreproducible builds. However, there have been studies that address the importance of reproducible builds. For example, Wheeler [45] describes a practical technique named diverse double compiling. By compiling the source files twice with different compilers, and verifying the compiled binaries, certain types of malicious attacks can be detected and prevented. According to Debian's documentation, this work partially motivates the reproducible builds practice [15]. Holler et al. [26] investigate the diverse compilation under embedded system, and experimentally quantify the efficiency of diverse compiling for software fault tolerance. Carnavalet and Mannan [25] conduct an empirical study, focusing on the reproducible builds in the context of security-critical software. Based on the experiments on the encryption tool TrueCrypt, they summarize the challenges of reproducibility in practice. Ruiz et al. [36] address the reproducibility in cloud computing. They adopt the term reconstructable software, and propose a prototype to simplify the creation of reliable distributed software.

In this study, we focus on the localization task for unreproducible builds, which has not been addressed in the existing studies.

7 CONCLUSIONS

In this study, we investigate the localization task for unreproducible builds. We present components that consider heuristic knowledge, similarity based information, as well as their integration as RepLoc. For empirical validation, we create four categories of publicly available datasets with 671 unreproducible packages from Debian. Extensive experiments reveal that RepLoc is able to effectively localize the files that lead to unreproducible builds. Furthermore, with the help of RepLoc, we successfully identified and fixed 6 new unreproducible packages from Debian and Guix.

For the future work, we are interested in the localization of problematic files for the tool-chain related issues. Also, inspired by the record-and-play techniques [34] from the crash reproduction based debugging research [27, 47], it would be interesting to leverage these techniques to detect more accurate correspondence between the build commands executed and the built binaries.

ACKNOWLEDGEMENTS

This work is supported in part by the National Natural Science Foundation of China under Grants 61772107, 61722202, 61502345, and 61403057, and in part by the Fundamental Research Funds for the Central Universities under Grant DUT16RC(4)62.

REFERENCES

[1] 2015. Protecting our customers from XcodeGhost. https://www.fireeye.com/blog/executive-perspective/2015/09/protecting_our_custo.html. (September 2015).

[2] 2017. Debian bug report logs – #773916: libical. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=773916. (August 2017).

[3] 2017. Debian bug report logs – #854293: manpages-tr. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=854293. (August 2017).

[4] 2017. Debian bug report logs – #854294: regina-rexx. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=854294. (August 2017).

[5] 2017. Debian bug report logs – #854362: fonts-uralic. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=854362. (August 2017).

[6] 2017. Debian packaging/source package. https://wiki.debian.org/Packaging/SourcePackage. (February 2017).

[7] 2017. Fixing a toolchain package. https://reproducible-builds.org/contribute/. (January 2017).

[8] 2017. GNU bug report logs – #28015: djvulibre. https://debbugs.gnu.org/cgi/bugreport.cgi?bug=28015. (August 2017).

[9] 2017. GNU bug report logs – #28016: libjpeg-turbo. https://debbugs.gnu.org/cgi/bugreport.cgi?bug=28016. (August 2017).

[10] 2017. GNU bug report logs – #28017: skalibs. https://debbugs.gnu.org/cgi/bugreport.cgi?bug=28017. (August 2017).

[11] 2017. The Guix System Distribution. https://www.gnu.org/software/guix/. (August 2017).

[12] 2017. Known issues related to reproducible builds. https://tests.reproducible-builds.org/index_issues.html. (July 2017).

[13] 2017. Notes on build reproducibility of Debian packages. https://anonscm.debian.org/git/reproducible/notes.git. (August 2017).

[14] 2017. Overview of reproducible builds for packages in unstable for amd64. https://tests.reproducible-builds.org/debian/unstable/index_suite_amd64_stats.html. (August 2017).

[15] 2017. Reproducible builds. https://reproducible-builds.org/. (August 2017).

[16] 2017. Reproducible builds bugs filed. https://tests.reproducible-builds.org/debian/index_bugs.html. (August 2017).

[17] 2017. Reproducible Builds Experimental Toolchain. https://wiki.debian.org/ReproducibleBuilds/ExperimentalToolchain. (February 2017).

[18] 2017. Reproducible builds: week 54 in Stretch cycle. https://reproducible.alioth.debian.org/blog/posts/54/. (October 2017).

[19] 2017. Reproducible builds: who's involved. https://reproducible-builds.org/who/. (August 2017).

[20] 2017. Timestamps In PE Binaries. https://wiki.debian.org/ReproducibleBuilds/TimestampsInPEBinaries. (August 2017).

[21] 2017. Validating Your Version of Xcode. https://electricnews.fr/validating-your-version-of-xcode/. (August 2017).

[22] 2017. Variations introduced when testing Debian packages. https://tests.reproducible-builds.org/debian/index_variations.html. (August 2017).

[23] Ludovic Courtès. 2015. Reproducible builds: a means to an end. https://www.gnu.org/software/guix/news/reproducible-builds-a-means-to-an-end.html. (November 2015).

[24] Steven Davies, Marc Roper, and Murray Wood. 2012. Using bug report similarity to enhance bug localisation. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 125–134.

[25] Xavier de Carné de Carnavalet and Mohammad Mannan. 2014. Challenges and Implications of Verifiable Builds for Security-critical Open-source Software. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 16–25. DOI: http://dx.doi.org/10.1145/2664243.2664288

[26] Andrea Höller, Nermin Kajtazovic, Tobias Rauter, Kay Römer, and Christian Kreiner. 2015. Evaluation of diverse compiling for software-fault detection. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 531–536.

[27] Nima Honarmand and Josep Torrellas. 2014. Replay Debugging: Leveraging Record and Replay for Program Debugging. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 445–456. http://dl.acm.org/citation.cfm?id=2665671.2665737

[28] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential biases in bug localization: Do they matter?. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 803–814.

[29] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 476–481.

[30] Hang Li. 2014. Learning to rank for information retrieval and natural language processing. *Synthesis Lectures on Human Language Technologies* 7, 3 (2014), 1–121.

[31] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2010. Bug localization using latent Dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972–990.

[32] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, and others. 2008. *Introduction to information retrieval*. Vol. 1. Cambridge university press Cambridge.

[33] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. 33–44.

[34] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 377–389. http://dl.acm.org/citation.cfm?id=3154690.3154727

[35] Shivani Rao, Henry Medeiros, and Avinash Kak. 2015. Comparing Incremental Latent Semantic Analysis Algorithms for Efficient Retrieval from Software Libraries for Bug Localization. *ACM SIGSOFT Software Engineering Notes* 40, 1 (2015), 1–8.

[36] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. 2015. Reconstructable Software Appliances with Kameleon. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 80–89. DOI: http://dx.doi.org/10.1145/2723872.2723883

[37] Davide Di Ruscio and Patrizio Pelliccione. 2014. Simulating upgrades of complex systems: The case of Free and Open Source Software. *Information and Software Technology* 56, 4 (2014), 438 – 462. DOI: http://dx.doi.org/https://doi.org/10.1016/j.infsof.2014.01.006

[38] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 345–355.

[39] Bunyamin Sisman and Avinash C Kak. 2012. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 50–59.

[40] Chakkrit Tantithamthavorn, Akinori Ihara, and Ken-ichi Matsumoto. 2013. Using co-change histories to improve bug localization performance. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*. IEEE, 543–548.

[41] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763.

[42] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 1–11. DOI: http://dx.doi.org/10.1145/2771783.2771797

[43] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 53–63.

[44] Shaowei Wang, David Lo, and Julia Lawall. 2014. Compositional vector space models for improved bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 171–180.

[45] David A Wheeler. 2005. Countering trusting trust through diverse double-compiling. In *Computer Security Applications Conference, 21st Annual*. IEEE, 13–pp.

[46] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 181–190.

[47] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 910–913. DOI: http://dx.doi.org/10.1145/2786805.2803206

[48] 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.

[49] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 14–24.