Options Matter: Documenting and Fixing Non-Reproducible Builds in Highly-Configurable Systems

Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, Mathieu Acher

Options Matter: Documenting and Fixing Non-Reproducible Builds in Highly-Configurable Systems

Georges Aaron Randrianaina
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
georges-aaron.randrianaina@irisa.fr

Djamel Eddine Khelladi
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
djamel-eddine.khelladi@irisa.fr

Olivier Zendra
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
olivier.zendra@inria.fr

Mathieu Acher
Univ Rennes, CNRS, Inria, IRISA
Institut Universitaire de France (IUF)
UMR 6074, F-35000 Rennes, France
mathieu.acher@irisa.fr

ABSTRACT

A critical aspect of software development, *build reproducibility*, ensures the dependability, security, and maintainability of software systems. Although several factors, including the build environment, have been investigated in the context of non-reproducible builds, to the best of our knowledge the precise influence of *configuration options* in configurable systems has not been thoroughly investigated. This paper aims at filling this gap.

This paper thus proposes an approach to automatically identify configuration options causing non-reproducibility of builds. It begins by building a set of builds in order to detect non-reproducible ones through binary comparison. We then develop automated techniques that combine statistical learning with symbolic reasoning to analyze over 20,000 configuration options. Our methods are designed to both detect options causing non-reproducibility, and remedy non-reproducible configurations, two tasks that are challenging and costly to perform manually.

We evaluate our approach on three case studies, namely Toybox, Busybox, and Linux, analyzing more than 2,000 configurations for each of them. Toybox and Busybox come exempt from non-reproducibility. In contrast, 47% of Linux configurations lead to non-reproducible builds. The approach we propose in this paper is capable of identifying 10 configuration options that caused this non-reproducibility. When confronted to the Linux documentation, none of these are documented as non-reproducible. Thus, our identified non-reproducible configuration options are novel knowledge and constitute a direct, actionable information improvement for the Linux community. Finally, we demonstrate that our methodology effectively identifies a set of undesirable option values, enabling the enhancement and expansion of the Linux kernel documentation while automatically rectifying 96% of encountered non-reproducible builds.

MSR '24, April 15–16, 2024, Lisbon, Portugal

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Software system models.*

KEYWORDS

Reproducible Build, Build System, Highly-configurable System

ACM Reference Format:
Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. 2024. Options Matter: Documenting and Fixing Non-Reproducible Builds in Highly-Configurable Systems. In *21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3643991.3644913

1 INTRODUCTION

Reproducibility, a fundamental principle in both scientific and technical fields, ensures that results can be consistently replicated under similar conditions. In software development, build reproducibility specifically refers to the ability to recreate a software build from its source code identically at different times and in varying environments. This approach is crucial for maintaining the integrity and reliability of software, as it enables developers to detect and address unintended changes, vulnerabilities, and ensure consistent behavior or performance across various platforms.

Non-reproducible builds in software development can arise from various factors: differences in timestamps, locale settings, file system ordering, build paths, uninitialized memory, and the degree of parallelism in the build process [18, 27, 30–32, 35]. Additionally, variations in the toolchain, changes in dependencies, and environment variables also significantly contribute to differences in the build outputs. Although such factors have been investigated, there is potentially another source of non-reproducibility that to the best of our knowledge has not been thoroughly explored, namely: the *configuration options* of configurable systems.

At first glance, options are simply used to enable or disable features or to have an effect on non-functional properties of a system. There is no reason to fear that they could affect the reproducibility of builds. Given a configuration, developers expect that the build process will produce the same output, regardless of the options selected once the build environment has been set and the build

system made deterministic. However, some informal discussions and observations suggest that some options can affect the build process itself, and thus the reproducibility of builds. For instance, the documentation of the Linux kernel on this topic describes only six configuration options that may have an impact. Unfortunately, the knowledge is most likely incomplete. There is no comprehensive, centralized, actionable list of options that can be leveraged to either inform developers about problematic options, prevent non-reproducible builds in the first place, or fix configurations subject to non-reproducibility. The key research question is: *Which configuration options, if any, have an impact on the reproducibility of builds?*

To the best of our knowledge, this question has not been thoroughly investigated in the academic literature or in real-world settings. A practical reason is that the number of configuration options in configurable systems is often very large. Developers usually test a few configurations (e.g., default configurations), and thus, may not be aware of the impact of options on the reproducibility of builds. For instance, the Linux kernel has more than 18,000 configuration options, and Busybox and Toybox, two other highly-configurable systems, have respectively 1,093 and 341 options. These lead to enormous sets of possible configurations to build. Considering these options as Boolean variables, the configuration spaces of Linux, Busybox, and Toybox have $\approx 10^{5000}$, $\approx 10^{300}$ and $\approx 10^{100}$ configurations. At this scale, it is tedious, time consuming and costly, hence difficult, to manually test and identify the (combinations of) options that affect the reproducibility of builds.

This paper proposes an approach to automatically identify configuration options causing non-reproducibility. It begins by building a set of builds in order to detect non-reproducible ones through binary comparison. We then develop automated techniques that combine statistical learning with symbolic reasoning to analyze thousands of configuration options of highly-configurable systems. Our methods are designed to both detect options contributing to non-reproducibility, and remedy non-reproducible configurations, two tasks that are challenging and costly to perform manually. We then iterate with additional exploration of the configuration space to validate the effect of configuration options causing non-reproducibility and to learn additional ones.

We evaluate our approach on three case studies, namely Toybox, Busybox, and Linux. Our first contribution is to assess the extent to which the configurations of these systems lead to non-reproducible builds, by analysing 2,000 configurations for each of them, 6,000 in total. Toybox and Busybox appear exempt from this issue, while 47% of the configurations of Linux lead to non-reproducible builds.

On Linux, our proposed approach was capable of identifying 10 configuration options that caused the non-reproducibility of those builds.

In the Linux documentation, none of these are are documented as non-reproducible. Thus, our identified non-reproducible configuration options are novel knowledge and constitute a direct, actionable information improvement for the Linux community. Finally, we demonstrate that our methodology effectively identifies a set of undesirable option values, enabling the enhancement and expansion of the Linux kernel documentation, while automatically rectifying 96% of encountered non-reproducible builds.

The remainder of the paper is organized as follows. Section 2 provides background on reproducible builds and how non reproducibility can be observed on a build of the Linux kernel. Related work is presented in Section 3. Section 4 explains our automatic approach using a classification algorithm and proposing exploration and fixing non-reproducible builds. Section 5 presents our research questions and experimental setup, leading to the results of Section 6. After discussing threats to validity in Section 7, we conclude in Section 8.

2 BACKGROUND

In this section, we present background on reproducible builds and build workflow, and how both relate.

2.1 Reproducible builds and environment

A build is considered *reproducible* when, given identical source code and build environment, two builds yield artifacts that are identical bit-by-bit. This encompasses intermediate files, and particularly the target binary.

It is well-known that embedding timestamps in build artifacts is a bad practice. In fact, two builds of the same source code in the same environment can differ only because they were build few moments apart.

Let us take the Linux kernel to build as an example. The Linux kernel is built with its build system called Kconfig. Kconfig relies on makefiles to run the builds once it is configured with the needed options. The bad practice presented above is observed in the Linux kernel. For instance, for the same configuration (tinyconfig here) built in the same environment, two binaries are obtained that differ because of the timestamps embedded in the binary, as depicted by the excerpt of diff produced by the diffoscope[1] tool in Listing 1.

```
  0xc10ce260 00000023 31204d6f 6e204f63 74203920 ...#1 Mon Oct 9
- 0xc10ce270 31343a32 373a3139 20434553 54203230 14:27:19 CEST 20
+ 0xc10ce270 31343a32 373a3336 20434553 54203230 14:27:36 CEST 20
  0xc10ce280 32330000 00000000 00000000 00000000 23..............
```

Listing 1: Build timestamp diff of two binaries of the same Linux configuration

This slight difference can impact other values in the binary such as the *BuildID*. The BuildID is a hash code computed from some parts of the binary. The difference on the date impacts it, as shown in Listing 2.

```
- GNU 0x00000014 NT_GNU_BUILD_ID (unique build ID bitstring)
Build ID: 21680dacce87ce3da3daac0f955db5775119416d
+ GNU 0x00000014 NT_GNU_BUILD_ID (unique build ID bitstring)
Build ID: 684826aea7385b7c8070eb948fb69e33de2d4ad6
```

Listing 2: BuildID diff induced by build timestamp diff

As previously mentioned, this embedding of metadata is documented and can be bypassed by overriding an environment variable and setting it to a fixed value. The Linux has other such metadata embedded through the build process, and we explain later in Section 5.3.2 how we bypass them. The Reproducible Builds[2] initiative Documentation offers more details about other kinds of such factors.

[1] https://diffoscope.org/
[2] https://reproducible-builds.org/docs/

2.2 Reproducible builds and configurations

Configurable systems provide configurations options for the developer or use to fit their needs. We present an excerpt of a configuration file for the Linux kernel in Listing 3 where the option for the `X86_64` architecture is enabled, as well as the option `MODULE_SIG_ALL`. Option `MODULE_SIG_FORCE` is disabled.

```
CONFIG_X86_64=y
...
# CONFIG_MODULE_SIG_FORCE is not set
CONFIG_MODULE_SIG_ALL=y
```

Listing 3: Few lines from a Linux configuration file

These configuration options constitute an additional layer to manage for reproducible builds. In fact, the Linux kernel documentation mentions few configuration options that could alter the build artifact. `MODULE_SIG_ALL` is part of the set of options managing the signing facilities of Linux kernel modules. When enabled, it automatically signs all modules during their build. It uses cryptographic keys generated at build time as depicted in Listing 4 that explicitly states "Build time autogenerated kernel key" followed by the actual key. A different key is generated for each build leading to different binaries. We show in Listing 4 the diff of the first bytes of the key in two binaries obtained from two different builds of the same configuration.Furthermore, the differences are not clustered at a single location, but scattered at several places within the artifacts. Hence, Listing 4 or Listing 1 are just excerpts of a difference of information that can lead to thousands of lines to review and analyze. Ultimately, no clear connection exists between the differences and configuration options, making it difficult, time-intensive, and error-prone to determine the exact options responsible. Overall, this knowledge is built on personal experience of Linux developers and maintainers, without any systematic approach or method to explore and identify configuration options that cause non-reproducibility.

```
  0xffffffff838c3400 06035504 030c2342 75696c64 2074696d ..U...#Build tim
  0xffffffff838c3410 65206175 746f6765 6e657261 74656420 e autogenerated
  0xffffffff838c3420 6b65726e 656c206b 65793020 170d3233 kernel key0 ..23
- 0xffffffff838c3430 31313036 31343334 35305a18 0f323132 1106143450Z..212
+ 0xffffffff838c3430 31313036 31343335 32315a18 0f323132 1106143521Z..212
```

Listing 4: Key signature diff among two binaries of the same configuration (excerpt)

While non-reproducibility due to build environment variation is well known (see Section 3), there is a striking lack of literature on the non-reproducibility due to configuration options. Our work aims to provide more insight on it through our approach and empirical study.

3 RELATED WORKS

Reproducible builds. Software builds, encompassing the related topic of reproducible builds, have been subject to a large body of work in the past few years [1, 4, 5, 8, 9, 14, 17, 21, 24, 25, 27, 34, 35, 38]. Lamb and Zacchiroli provide a general overview and a problem statement of reproducible build [18]. They notably report on insights of the Reproducible Builds project making the Debian Linux distribution almost reproducible. As of August 2023[3], around 90% of the Debian packages are reproducible, depending on architectures and branches. Other major open-source projects have followed the same path.

We concur with the general definition given by Lamb and Zacchiroli [18]: *"The build process of a software product is reproducible if, after designating a specific version of its source code and all of its build dependencies, every build produces bit-for-bit identical artifacts, no matter the environment in which the build is performed."*. However, we remark that configuration options, and specifically compile-time options, can lead to a new *variant* of the software source code.

It is important to distinguish between "variant" and "version" (such as commit or release) when considering the reproducibility of builds. Options can have an impact on the build process, as they can result in a new source code that needs to be built with the same environment. However, the variant dimension has been largely neglected in the past, and both versions and variants should be considered when assessing the reproducibility of builds.

To the best of our knowledge, the literature on reproducible builds does not consider the impact of configuration options. For instance, Lamb and Zacchiroli did not mention the impact of configurations and configuration options as part of their tour on build reproducibility [18].

Locating and fixing reproducibility. The detection of differences between builds can be easily achieved by comparing the checksums of their artifacts. This process is straightforward. However, it can be extremely difficult to understand the root cause of the differences. Tools such as diffoscope can help in this task by providing a detailed comparison of the built artifacts. We employed diffoscope in our study to compare the artifacts of the builds. As elaborated in Section 2.2, managing the large volume of differences information can quickly become an overwhelming task, especially since this effort must be repeated across various configurations. Furthermore, pinpointing which specific option is causing these differences presents an additional challenge.

BuildDiff can extract and classify build changes with a high accuracy [19]. RepLoc aims at localizing the problematic files for unreproducible builds, searching for the most relevant build scripts to the inconsistent artifacts [30]. Ren et al. [31] propose RepTrace, a framework that exploits the uniform interfaces of system call tracing. RepTrace can identify the root causes for unreproducible builds by analyzing and monitoring the executed build commands. As a follow-up, Ren et al. [32] propose RepFix, a tool that can automatically fix the root causes of unreproducible builds. Bajaj et al. [3] conducted an empirical study to investigate the transition of builds from reproducible to non-reproducible and vice versa over Debian and Arch Linux packages and identified 16 root causes of non-reproducible builds.

These related works do not consider the impact of configuration options on the build process, and it is unclear how traces and build scripts relate to the configuration options. Furthermore, at the scale of the Linux kernel, the number of system calls and build commands can be very large, which makes trace analysis challenging.

Configurable systems. A connection between reproducibility and quality assurance exists. Hence, some testing methods can be somehow repurposed to target reproducibility issues. The question is how to adapt them to the context of reproducible builds and highly-configurable systems. In software product line engineering, a large body of work demonstrates the need for configuration-aware

[3] https://tests.reproducible-builds.org/debian/reproducible.html

testing techniques and proposes methods to sample and prioritize the configuration space [7, 12, 15, 16, 22, 26, 33, 37, 39]. In [36], the iTree algorithm is proposed to efficiently cover configurations using machine learning. Gazzillo *et al.* [11] challenge the community to locate configurations at a given point of interest. Zhang and Ernst [40] propose an algorithm to diagnose crashing errors related to software misconfigurations. Halin *et al.* [13] exhaustively test all possible 26,000+ configurations of an industry-strength, open source configurable software system, JHipster. They found that 35.70% configurations fail and they identify the interactions that cause the errors using association rules and logs errors. Randrianaina *et al.* [28, 29] build configurations in an incremental way with the ambition of speeding up the build process. They observed that, in some settings, certain binaries differ, without delving into the identification of the specific options responsible for these differences. It is also unclear whether the discrepancies were due to incremental build (as opposed to traditional builds) or configuration options. Chen et al. [6] propose a method to diagnose configuration errors with regard to performance.

However, non-reproducible builds do not manifest as crash, errors, or performance drops, but as *differences in the artifacts*. These differences information may not be as informative as a crash or a performance drop, can be scattered across the artifacts, and can be difficult to trace back to individual options or combinations of options. Hence, two problems arise: analyzing differences, and finding the causes from this information.

A unique challenge is to identify the configuration options (among thousands of options in a large configuration space) that are responsible for the differences in the artifacts. We are unaware of any work that thoroughly studies the impact of configurations options on reproducibility or that addresses this challenge.

4 APPROACH

Our approach is based on two principles: (1) statistical learning to infer and identify which option values are causing reproducibility issues; (2) symbolic reasoning to further explore related problematic options and fix configurations without compromising their logical constraints. A major benefit of this automated approach is that it eliminates the necessity to manually scrutinize differences between numerous builds.

This section outlines our strategy identifying configuration options contributing to the non-reproducibility of builds. The approach comprises three primary stages, as illustrated in Figure 1. In the initial stage (①, detailed in Section 4.1.2), a configuration is built and is examined for the reproducibility of its build. Then, both the configuration and the reproducibility test result are encoded and stored in a tabular dataset utilized during the learning phase in (②, detailed in Section 4.2). The learning phase yields a list of options strongly correlated with the non-reproducibility of builds. This list is further enriched through an exploration of the dependencies associated with the given options in (③, detailed in Section 4.3). Finally, the list is used to fix configurations to make them reproducible (④, detailed in Section 4.3.1) and add the identified options to the documentation (⑤, detailed in Section 4.3.2).

4.1 Building and reproducibility check

4.1.1 Building configurations. We pursue two goals. The first is to empirically verify whether configurations lead to reproducible builds. The second is to test as many options as possible, in different configurations, and hopefully learn patterns of interest that can explain non-reproducibility. In both cases, we need to build numerous configurations.

Our approach first generates random configurations for the target software. Standard configurations exist (like "tinyconfig", "defconfig", or "allyesconfig") but they yield to low diversity. We prefer to start with a "randconfig", where the values of options are set in a random manner. The *randconfig* tool is widely used in the Linux community to test the kernel in different settings [2, 11, 20, 23]. The tool is actually supported in many KConfig-based projects such as Toybox or Buysbox. Utilizing random generation serves to diversify the set of activated and deactivated configuration options. A learning procedure can then infer which option values have an impact on reproducibility (see next subsection).

4.1.2 Build environment and reproducibility. All configurations are built inside a container, that contains the same build environment, with the necessary utilities for the builds to succeed, and also environment variables whose values are fixed. Fixing these makes it possible to focus on the impact of *configuration options* on the reproducibility. The list of environment variations that could impact the build reproducibility is retrieved from official documentations, mailing lists or available public repositories explicitly mentioning reproducibility of the given software build. To determine whether a configuration build is reproducible, the same configuration is built twice in two separate runs of the aforementioned container; a build is considered reproducible if and only if the produced binaries are bit-by-bit identical. Some configuration builds can also fail; for the remainder of the paper, only configurations that successfully build are retained for analysis.

4.2 Learning

4.2.1 Data encoding and preprocessing. The result of the previous step is represented in a tabular format.

We mainly focus on configuration options capable of enabling or disabling real features. These options are encoded in Kconfig as boolean or tristate (boolean with an extra possible value `m` to enable an option as a module). Hence, we do not consider other category of options (string, hexadecimal, integer) that are just parameters for real features and can be set by default.

Each configuration option represents a column and a predictive variable as depicted in Figure 1 following a specific encoding: `0` for a disabled option, `1` for an enabled option and `2` for an option enabled as a module. Hence, a row represents a configuration. A last column keeps the information on whether the build was reproducible or not (the binary class of a configuration). Before the learning phase, our tabular data is cleaned up. All configuration options that have constant values (e.g., always set to `yes`) are removed, since they do not contribute information.

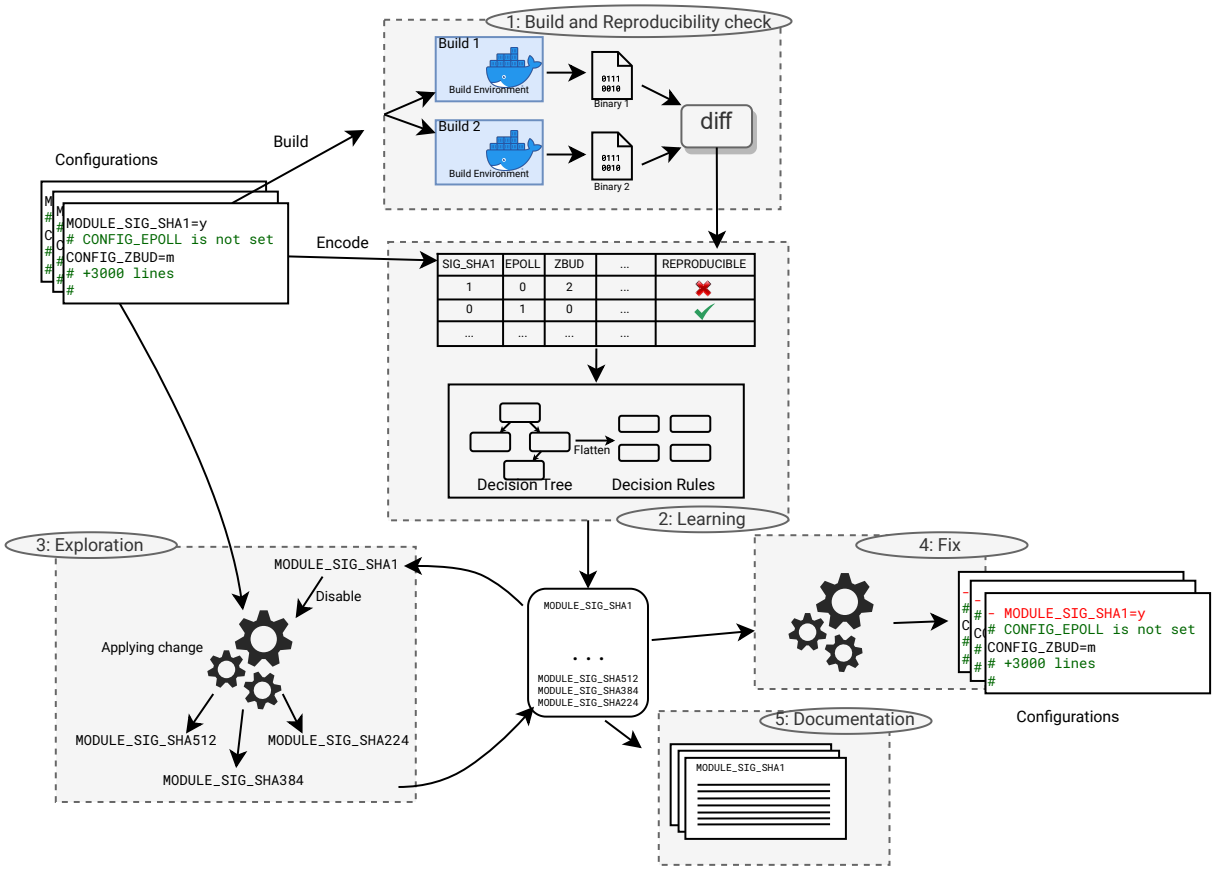

Figure 1: Approach overview

4.2.2 Learning phase. The learning phase uses this tabular data as input. We aim to resolve a classification problem by determining whether a configuration and its build is reproducible or non-reproducible, based on the specific configuration options used. We thus choose to rely on decision trees. The principle of a decision tree algorithm involves creating a model that predicts the value of a target variable (here: reproducible or not) by learning simple decision rules. This method is particularly suited for our context of identifying non-reproducible builds because decision trees are highly interpretable. They provide a clear visualization of the decision-making process, enabling developers to pinpoint and address the specific (combinations of) options contributing to non-reproducible builds.

We call such options *Configuration Options causing Non Reproducibility ("CO-NR")*. Figure 4 shows an example of a decision tree obtained on Linux configurations. This tree structure is then flattened, in order to obtain Decision Rules with independent CO-NR, that will be considered in the subsequent exploration phase. For example, in Figure 4 when the option `MODULE_SIG_SHA1` is activated, it leads to 328 CO-NR. When it is deactivated and 1) `GCOV_PROFILE_FTRACE` is activated, it leads to 95 non-reproducible builds or 2) `DEBUG_INFO_REDUCED` is activated, it leads to 33 non-reproducible builds.

4.3 Exploration

The Exploration phase consists in identifying the *siblings* of the options we have already identified. A sibling is simply a possible alternative of the identified option. For example, `MODULE_SIG_SHA1` is a choice among multiple hash algorithms. Hence, regardless of the choice of the hash algorithm, the module signing key will be written anyway. Thus the impact in terms of reproducibility remains the same. We also identify their parent which permits to remove a whole range of suspected configuration options.

We use the tool ConfigFix [10] to extract the set of choices at the same level of a given option. ConfigFix is a tool able to produce fixes for configuration conflicts. For example, if a user wants to enable an option without knowledge of its dependencies, ConfigFix can generate a series of diagnoses, each representing a list of options and their corresponding values required to apply the desired change.

We present the exploration strategy in Algorithm 1. Given a set of configurations which are not reproducible and the set of identified options, we check which options from the obtained list are present in each configuration that was not reproducible. We build a set with the option and its value to no in Line 8 and apply the set of changes to the configuration using ConfigFix. This gives

us a list of diagnoses from which we will extract the alternatives of the option that is deactivated.

We illustrate the extraction of the sibling from a diagnosis of ConfigFix with an example. Let us consider that we want to deactivate `MODULE_SIG_SHA1`. We feed ConfigFix with the option name and the desired value, which is `no` since we want to deactivate it. We obtain a set of diagnoses from which we can observe the following patterns. A diagnosis with two options with their respective value `MODULE_SIG_SHA1` set to `no` and another hash algorithm, say `MODULE_SIG_SHA512` set to `yes`. Or `MODULE_SIG_SHA1` set to `no` with another hash algorithm enabled, say `MODULE_SIG_SHA384`. The size of diagnosis of this kind is always of two and contains the initial option deactivated and another alternative set to `yes`. Hence the filter of diagnosis of size 2 from all diagnoses in Line 11 of Algorithm 1 which extracts for us the name of the alternative to the initial option.

We leverage the structure of the name of a Kconfig option to heuristically extract its parent. We explain how the function *getParent*() in Line 16 is implemented. For a given option `MODULE_SIG_SHA1`, removing the last part `_SHA1` gives us `MODULE_SIG` which is in fact the parent of the initial option and necessary dependency to enable it. The function returns an option if and only if the new modified name exists.

Finally, the set returned by Algorithm 1 is a richer version of the configuration options selected by the decision tree.

Algorithm 1: Exploration Phase Algorithm

```
   input  : Set of configurations C, List of options O
   output : Augmented list of options
1  begin
2      OutSet ← {}
3      foreach c ∈ C do
4          change ← {}
5          foreach o ∈ O do
6              if o is enabled in c then
7                  Add o to OutSet
8                  Add (o,no) to change
9              diagnoses ← getDiagnoses(c, change)
10             foreach diag ∈ diagnoses do
11                 if size(diag) == 2 then
12                     rest ← diag \ change
13                     if size(rest) == 1 then
14                         opt ← first(rest)
15                         Add opt to OutSet
16                         Add getParent(opt) to OutSet
18     return OutSet
```

4.3.1 Fixing. The aim of the Fixing phase is to get rid of CO-NR in each configuration. If they are enabled, we deactivate all the CO-NR. In order to do it properly, we use ConfigFix presented in Section 4.3 to get a diagnosis. While we focused only on the dependency information provided before, we apply the fix to generate a real configuration file with the needed modifications. To evaluate how precise the list is, we build again the configurations for which the builds were not reproducible.

4.3.2 Documentation. This phase seeks to enhance the documentation of the target software by incorporating — if not already included — the identified options into the project's reproducible build documentation .

5 EVALUATION METHODOLOGY

This section describes the evaluation of the approach presented in Section 4. We first explain our research questions. Then we present the protocol we followed in order to answer them. Finally, we describe our experimental setup.

5.1 Research Questions

We define the following research questions (RQ):

RQ1: To what extent do configurations lead to non-reproducible builds? The goal of this research question is to determine whether changing configurations of a given software produces non-reproducible builds. To answer this question, we first randomly generate distinct configurations of the software, then build them twice in two different containers with the same build environment.

RQ2: Can we identify configuration options that cause non-reproducible builds? Given configurations that are not reproducible, we aim to identify exactly which configuration options caused this. We divide this research question in three more precise ones:

- *RQ2.1: How accurately does the learning phase identify configuration options causing non-reproducible builds?* To answer this research question, we evaluate the accuracy of the classification algorithm, varying the size of the testing and training sets.
- *RQ2.2: Are the identified configuration options novel compared to existing documentation?* Given the list obtained by our approach, we check in existing official documentation (if any) whether these options are explicitly listed.
- *RQ2.3: What kind of configuration options are causing non-reproducible builds?* This consists in understanding the options from the explanations provided in the code documentation.

RQ3 – Can we fix non-reproducible builds? We evaluate the precision of our detection by disabling the identified CO-NR options and rebuilding the modified configuration. The accuracy depends on whether the modification has now made the build reproducible.

5.2 Case studies

Table 1 details the selected case studies [4]. All our dataset and results are given in our companion web page [5]. We chose three configurable systems, namely Linux, BusyBox, and ToyBox. They vary in size from 69k LOC to 21M LOC, and number of configuration options from 341 to 18637.

[4] LoC computed with `cloc` — https://github.com/AlDanial/cloc
[5] anonymous.4open.science/r/msr24repro-B186/

System	Version	LoC	#Options
Linux	5.13	21605254	18637
Busybox	1.36.1	220304	1078
Toybox	0.8.5	69355	330

Table 1: Studied systems

5.3 Evaluation Protocol

5.3.1 Configurations.

Random configurations. Each of our use-cases relies on the Kconfig build system. Kconfig provides a command to generate a random configuration; this is the one we use. For each use-case, 2000 random configurations are generated. For each new configuration, all comments are removed, then the remaining content is sorted alphabetically and hashed. This hash is then compared to the hashes of the previous configuration, to ensure that each configuration is present only once.

Preset. Configurations for the Linux kernel were generated with a preset for `x86_64` architecture. For Busybox, we used its custom script written by its developers[6] that calls Kconfig's randconfig and additional hardcoded modifications in order to get valid configurations. For Toybox, we wrote a script that calls Kconfig's randconfig but removes some unimplemented features since they can cause build failure.

5.3.2 Build Environment. All possible variations due to the build environment must be removed, in order to only focus on investigating the impact of *configurations* on the reproducibility of the build of a configurable software. Thus, some environment variables need to be pre-set, from various sources detailed hereafter.

Linux Kernel. The official documentation on Reproducible Builds[7] lists few environment variables that could be pre-set in order to ensure reproducible builds. In our experiments, several variables are overridden to get a fixed build environment. `KBUILD_BUILD_TIMESTAMP`, that embeds the build date into the binary is set to "`Sun Jan 1 01:00:00 UTC 2023`" for every build. Otherwise, building the same configuration twice two seconds apart is enough to make the build non-reproducible. `KBUILD_BUILD_USER=user` and `KBUILD_BUILD_HOST=host` define the string "`user@host`" displayed during boot and in `/proc/version` of Linux; they can vary because they depend on the output of commands `whoami` and `host` respectively. Last but not least, a variable that is pre-set was not found in the official documentation but in the mailing list[8]: `KBUILD_BUILD_VERSION="1"` is a counter that increments every time a build is performed in the same current directory, its value being then embedded inside the produced binary. Setting this variable is only necessary when performing incremental builds in the same directory; however we keep it set to 1 to avoid anything that could alter the produced binary.

Busybox and Toybox. Busybox website documentation section contains a Q&A and help for the commands it provides. The Q&A mentions a possible difference between two different binaries for the same configuration due to *"different compilers/linkers used, because of build timestamp and so on"*[9]. However, the mention of the variable to set in order to remove the timestamp variation was found in the mailing list from 2017[10]. Hence setting `KCONFIG_NOTIMESTAMP=1` disables the build timestamp embedding in the binary. On the other hand, we did not find any explicit timestamp embedded in the binary. Thus, we do not pre-set anything for Toybox.

Our build environment is particularly inspired by Linux tooling to massively build Linux kernel such as KernelCI[11] or TuxMake[12]. In fact, our Docker image is derived from TuxMake's to build Linux kernel with GCC 10.3. Since these Docker images are updated often, we fixed our build environment to the tag `20230912`, i.e its version of September 12th 2023. All experiments ran on a server with Debian 12 (Bookworm) with an AMD Epyc 7532 processor and 512 GB of RAM.

Classification Algorithm. We use the decision tree classifier provided by *scikit-learn*[13] version 1.3.1. We rely on default configuration to make the approach very practical: there is no need to perform hyperparameters tuning to already get good results.

6 RESULTS

This section presents our answers to the RQs of Section 5.1.

6.1 RQ1: To what extent do configurations lead to non-reproducible builds?

For each of the three systems, we build twice a set of 2,000 configurations. The rationale was to compare whether two builds of the same configuration lead to the same outcome.

The result of an individual build can be either a success or failure. Among the builds that succeed, the build can be either reproducible or not, depending on whether two builds of a same configuration produce binaries that are bit-by-bit identical or not.

Busybox. To pinpoint the source of non-reproducible builds, the typical workflow is to slightly vary the build environment. Changing the build path between two builds of the same configuration for Busybox impacts 49.75% of the configurations, causing their build to be non-reproducible (presented in Figure 2 under the name *Busybox (alter)*). The decision tree identifies the option involved which is `DEBUG`. In fact, this option includes some debug information in the binary including the build path. Thus, interactions exist between configuration options and build environment. This can be solved in two ways, either by disabling the option, or not changing the build environment. There, a trade-off is to be made because the developer may need `DEBUG` and require the build to be reproducible. Building in the same directory solved the issue and the configurations we have picked for Busybox are 100% reproducible as shown in Figure 2. *Overall, altering the build environment in Busybox identified*

[6] https://git.busybox.net/busybox/tree/scripts/randomtest?h=1_36_stable
[7] https://www.kernel.org/doc/html/latest/kbuild/reproducible-builds.html
[8] kbuild: add documentation of `KBUILD_BUILD_VERSION`, https://patchwork.kernel.org/project/linux-kbuild/patch/1428216268-3545-1-git-send-email-hofrat@osadl.org/
[9] https://busybox.net/FAQ.html, *"I want to use Busybox as part of the Linux-based firmware for a new device. Will it create any license issues in future ?"*
[10] http://lists.busybox.net/pipermail/busybox/2017-July/085590.html
[11] https://kernelci.org/
[12] https://tuxmake.org/
[13] https://scikit-learn.org

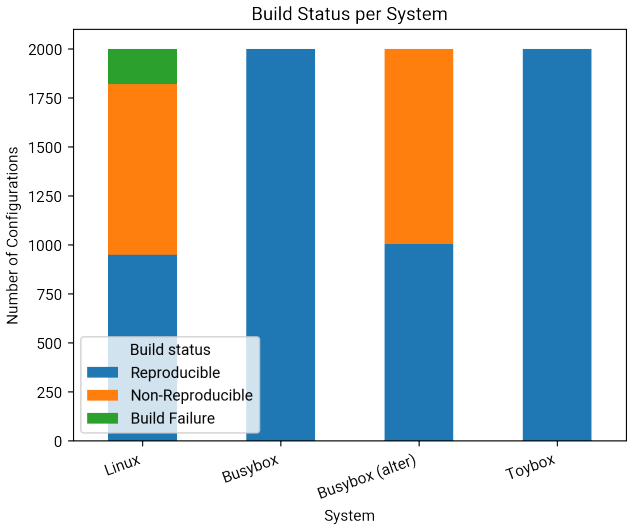

Figure 2: Build status per system

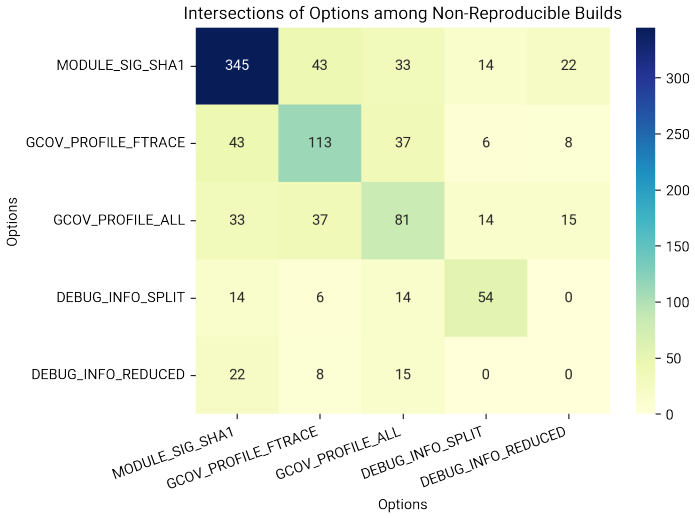

Figure 3: Intersections of Options from the Decision Tree

the DEBUG *option as key to achieving 100% reproducibility, either by disabling it or maintaining a consistent build path.*

Toybox. For Toybox, 99.95% of builds are reproducible, the remaining 0.05% (1 over 2000) failed. All successful builds were thus reproducible.

Linux kernel. For the Linux kernel 43.6% of builds were reproducible, 47.45% non-reproducible and 8.95% failed. We take care of using a consistent build path and environment in the first place.

reproducibility of the builds of a configurable software. From the studied systems, non-reproducibility happened only on Linux.

> ***RQ*$_1$:** Out of three case studies, we only observed in Linux 47.5% non-reproducible builds due to configuration options. For Linux, this is a huge rate and a crucial issue to address.

6.2 RQ2: Can we identify configuration options that cause non-reproducible builds?

The data from the initial steps of our approach provided a Decision Tree (see Section 4.2), a sample of which is presented in Figure 4 for the Linux kernel based on training set of 60% and 40% of the test set.

6.2.1 RQ2.1: How accurately does the learning phase identify configuration options causing non-reproducible builds? In Figure 5, we report on how the number of different configurations and builds used to train a model affects its accuracy. Accuracy is the fraction of predictions our model got right, both reproducible and non-reproducible. In the context of our model, the accuracy is given by the formula:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (1)$$

where TP (true positives) are configurations correctly identified as reproducible, TN (true negatives) are configurations correctly

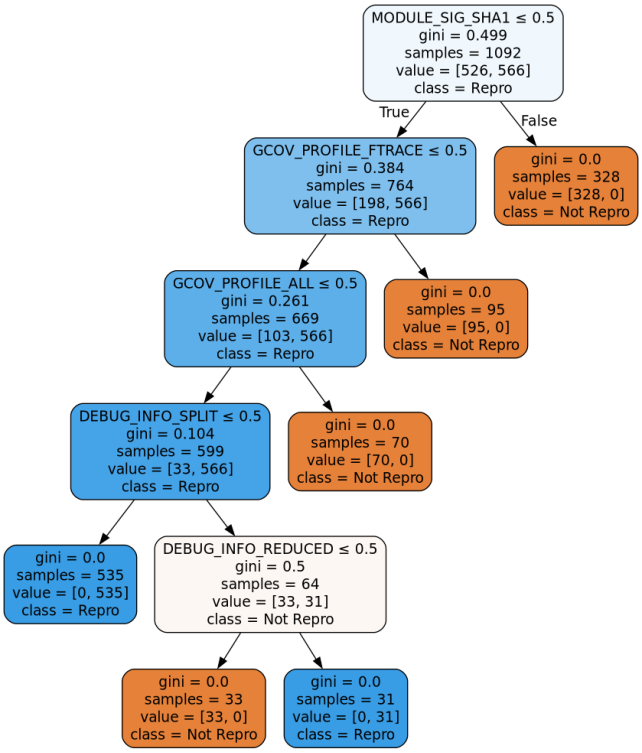

Figure 4: Decision tree obtained for Linux

identified as non-reproducible, FP (false positives) are reproducible configurations incorrectly labeled as non-reproducible, and FN (false negatives) are non-reproducible configurations incorrectly labeled as reproducible. We compute accuracy on the testing set, that consists of the whole dataset of 2,000 configurations without the configurations used in the training. We train our model with a

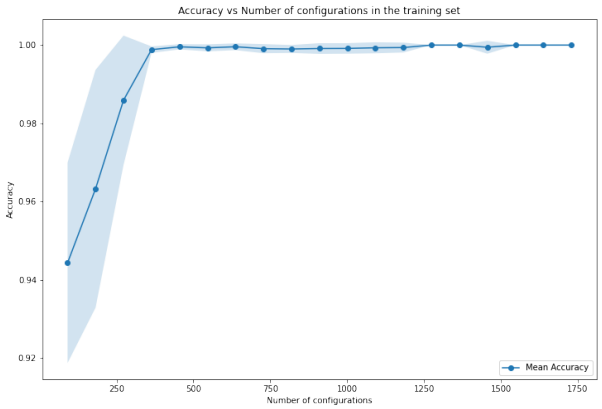

Figure 5: Accuracy with regard to the number of configurations and builds used as part of the training

decision tree, using sklearn version 0.23 and without tuning hyperparameters, relying on default values. We repeat the experiments 10 times.

The graph shows that as we increase the configurations from 100 onwards, the model accuracy improves quickly. By the time we reach 500 configurations (25% of the dataset), the accuracy is already high, leveling off just above 98%. With 60% of training set and 40% of test set, we reached an accuracy of 1 systematically. This quick rise in accuracy suggests that the model accurately learns from the first few hundred configurations. The shaded area around the accuracy line shows that there is less accuracy deviations in the model performance as we add more configurations. In simple terms, the model becomes consistent in its predictions. It is noteworthy that the trends of precision and recall are similar with 100% score and a few configurations and builds (see companion web page), thereby confirming our key findings concerning accuracy.

Regarding the model interpretability, Figure 4 shows that every configuration leading to non-reproducible builds includes a minimum of one option that is identified by the decision tree.

Moreover, Figure 3 shows the overall identified CO-NRs over all 1821 configurations built without errors. Each cell gives the number of non-reproducible builds involving the two configuration options in its line and column. With only 1821 built configurations for Linux, we were able to identify 10 options that are the direct cause of non-reproducible builds.

Therefore, we can answer this RQ2.1 and confirm that we can identify problematic options for the build reproducibility. The main takeaway is that our learning model gets to high accuracy with a relatively small number of configurations.

6.2.2 RQ2.2: Are the identified configuration options novel compared to existing documentation? We focus here on the official *Linux kernel documentation about reproducible builds*. None of the options shown in Table 2 detected by the presented approach appears in this documentation. Thus is of great impact for Linux developers to know about them. Our approach can be used to augment the documentation w.r.t. non-reproducible builds caused by problematic configuration options.

Table 2: Identified options and their category.

Option	Category
`MODULE_SIG_SHA1`	Module Signing
`MODULE_SIG_SHA224`	Module Signing
`MODULE_SIG_SHA256`	Module Signing
`MODULE_SIG_SHA384`	Module Signing
`MODULE_SIG_SHA512`	Module Signing
`MODULE_SIG`	Module Signing
`GCOV_PROFILE_FTRACE`	Profiling
`GCOV_PROFILE_ALL`	Profiling
`DEBUG_INFO_SPLIT`	Debug Info
`DEBUG_INFO_REDUCED`	Debug Info

6.2.3 RQ2.3: What kind of configuration options are causing non-reproducible builds? The Linux documentation on reproducible builds presents only 6 configuration options, grouped by category. The *Module Signing* section mentions 4 options, especially `MODULE_SIG_ALL` that generates a different temporary key for each build. The documentation provides a guideline on how to set other related configuration options.

The *Structure randomisation* section mentions the configuration option `RANDSTRUCT` that randomizes structure layout based on a seed. Though an option doing this very task exists in Linux using the GCC plugin by enabling `GCC_PLUGIN_RANDSTRUCT`, the option `RANDSTRUCT` does not exist in the version of the kernel we studied.

Finally, the section *Debug info conflicts* seems to emphasize the fact that some options can be "too reproducible". Indeed, it mentions that if the necessary variables for reproducible builds are set, a vDSO [14] debug information may be identical even for different kernel versions. Setting an arbitrary salt string to option `BUILD_SALT` should prevent this. However, we did not experience anything related to this issue in our experiments.

We reuse the categories of the documentation if the identified options fit in them. Indeed, most of the identified options can fit in either the category of "Module Signing" or "Debug Info". However, `GCOV_PROFILE_ALL` and `GCOV_PROFILE_FTRACE` are not mentioned anywhere in the documentation. These configuration options are used to profiling the Linux kernel. This should encourage documentating the impact of *Profiling* options on the reproducibility of builds.

To summarize, we found ten configuration options not explicitly listed in the Linux documentation and capable of explaining the 47.5% rate of non-reproducible builds. For the six options listed in the Linux documentation: (1) our approach does not identify the option `MODULE_SIG_ALL` but rather the parent `MODULE_SIG`, since the previous options are not the actual underlying issue; (2) the option `RANDSTRUCT` was non-existent in the version of Linux we use, and it is not the fault of our approach. Interestingly, it shows the necessity of continuously updating documentation. Overall, our approach is more precise and capable of identifying new configuration options.

[14] virtual Dynamic Shared Object (vDSO)

RQ_2: Our automatic approach accurately identified a list of novel configuration options that cause non-reproducibility and that are not documented in the Linux documentation, thus providing a significant novel result towards reproducible builds.

6.3 RQ3: Can we fix non-reproducible builds?

We now report the result of fixing the configurations for which the builds are not reproducible as presented in Figure 2. In fact, 47.45% (872) of the configurations builds of Linux were not reproducible, from our configuration set. The procedure to fix and generate new configuration as described in Section 4.3.1 using ConfigFix resulted in three different cases:

Case 1: No diagnosis. When ConfigFix does not find any solution to apply the provided modification, it simply returns without any diagnosis. This happens on 3/872 configurations for our fix, which represents less than 0.5% of the cases.

Case 2: Fixed reproducible builds. Over the non-reproducible builds, 838 configurations were fixed and allowed reproducible builds, which represents 96% of fixes.

Case 3: Non reproducible builds. 31 configurations representing 3.5% were not fixed. After investigation, we found out that even removing the parent of `MODULE_SIG_SHA1` which is `MODULE_SIG`, it was still activated by ConfigFix since another option `IMA_APPRAISE_MODSIG` was also sufficient to enable `MODULE_SIG_SHA1`. This shows the limits of our dependency extraction based on the textual representation of the options in Kconfig. Further improvements on the dependency extraction is left for future work.

Discussion. Our algorithm relies on *getParent* that selects parent options logically related to already identified CO-NRs. Our experiments show that our method is effective to fix many non-reproducible builds, but is incomplete. A possible improvement is to modify *getParent* to also select transitive options, possibly far in the Kconfig tree. However the risk exists to include too many options and thus deviate from the goal of maintaining a minimal set of changes. A possible remediation to this is to involve an expert (*e.g.,* Linux contributor) who will review the dependencies and thus manually set part of the knowledge on how to fix configurations – exactly as we did for understanding the 3% remaining configurations.

Busybox. We reported in Section 6.1 the existence of interactions between configuration options and the build path, which leads to 49.75% of non-reproducible builds. There are two alternatives to fix this reproducibility issue. Always perform a build in the same directory. Otherwise, remove the `DEBUG` option.

We reduced the amount of configurations with non-reproducible builds from 47.45% to 1.4% considering the initial campaign with 2000 builds of Linux.

RQ_3: Our approach could fix 96% of the non-reproducible builds thanks to our list of identified options in RQ2.

7 THREATS TO VALIDITY

We now discuss internal and external threats to validity.

7.1 Internal Validity

The first risk here is related to size of the data set of 2,000 configurations that limits us in the identification of options. However, our goal in this paper was not to identify a complete list of CO-NRs, including the six already known in the documentation. Our goal was to show with this empirical study that this problem is real and is systematic. Thus, experimenting on 2,000 configurations was enough to show evidence of non-reproducibility due to configuration options. Moreover, the cost of building and running our approach had to be taken into account: it took more than 60 hours of computation for gathering the training and testing set. Our comparison with the documentation further confirmed our novel empirical observations, i.e. that we did find 10 novel CO-NRs that were not documented as causing non-reproducibility. For completeness and a larger list of options, we plan to replicate our study on different sets of diverse configurations.

Finally, to remove the risks of misidentification of non-reproducible builds and options, we performed our experiment twice for each system. It means that in total, each configuration was built 4 times: twice to actually detect non-reproducible builds, and twice again as a second execution of our experiment. This allowed us to confirm the observed results and mitigate the risk of having an environmental factor influencing our observations of non-reproducibility.

7.2 External Validity

We conducted experiments on the Linux kernel, ToyBox, and BusyBox, which are highly complex, configurable C-based software systems using the Kconfig and Make build systems. Further experiments are necessary before generalizing the observed results to other build systems and other configurable software. We further should replicate on other sets of diverse configurations before generalizing our ability to systematically identify CO-NRs. Nonetheless, we showed in this paper that this was doable, and we also empirically demonstrated the existence of reproducibility issues due to options.

8 CONCLUSION

The issue of non-reproducible builds due to environment changes is known in the literature. In this paper we shed light on the *non-reproducible builds caused by configuration options*. We proposed an automatic approach to identify configuration options causing non-reproducibility. We then developed automated techniques that combine statistical learning with symbolic reasoning to analyze over 20,000 configuration options. We finally attempted to fix the non-reproducible builds thanks to the identified list of options.

Our empirical evaluation on three case studies, namely Toybox, Busybox, and Linux, relied on 2,000 configurations for each of them. Our results showed that Toybox and Busybox are exempt from this issue, whereas 47% of the Linux configurations lead to non-reproducible builds.

Our approach was capable of identifying 10 novel configuration options that caused the non-reproducibility of those builds. When

confronted to the Linux documentation, none of these are are documented as non-reproducible. Thus, our identified non-reproducible configuration options are novel knowledge and constitutes a direct, actionable information improvement for the Linux community. Finally, we demonstrated that our methodology effectively identifies a set of undesirable option values, enabling the enhancement and expansion of the Linux kernel documentation while automatically rectifying 96% of encountered non-reproducible builds.

As future work, we plan to replicate our experiment on others generated sets of random configurations. Since we found 10 novel options causing non-reproducible builds with only 2,000 configurations, we hope to identify more problematic options with larger and, more importantly, more diverse sets of configurations. Moreover, we believe our results have a positive impact on Linux developers. They lead to actionable actions: in the future, we plan to send a pull request to the Linux documentation with the list of identified options causing non-reproducibility of builds, and more generally engage discussions about this new issue and our approach.

REFERENCES

[1] *A fast, scalable, multi-language and extensible build system.*

[2] I. Abal, J. Melo, S. Stanciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski, *Variability bugs in highly configurable systems: A qualitative analysis*, ACM Trans. Softw. Eng. Methodol., 26 (2018), pp. 10:1–10:34.

[3] R. Bajaj, E. Fernandes, B. Adams, and A. Hassan, *Unreproducible builds: Time to fix, causes, and correlation with external ecosystem factors*, Empirical Software Engineering, 29 (2024).

[4] C.-P. Bezemer, S. McIntosh, B. Adams, D. M. German, and A. E. Hassan, *An Empirical Study of Unspecified Dependencies in Make-Based Build Systems*, Empirical Software Engineering, 22 (2017), p. 3117–3148.

[5] Q. Cao, R. Wen, and S. McIntosh, *Forecasting the duration of incremental build jobs*, in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 524–528.

[6] Z. Chen, P. Chen, P. Wang, G. Yu, Z. He, and G. Mai, *Diagconfig: Configuration diagnosis of performance violations in configurable software systems*, in Proceedings of the 2023 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023.

[7] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter, *Feedback driven adaptive combinatorial testing*, in Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, New York, NY, USA, 2011, ACM, pp. 243–253.

[8] S. Erdweg, M. Lichter, and M. Weiel, *A sound and optimal incremental build system with dynamic dependencies*, ACM Sigplan Notices, 50 (2015), pp. 89–106.

[9] S. I. Feldman, *Make — a program for maintaining computer programs*, Software: Practice and Experience, 9 (1979), pp. 255–265.

[10] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev, *Configfix: Interactive configuration conflict resolution for the linux kernel*, in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2021, pp. 91–100.

[11] P. Gazzillo, U. Koc, T. Nguyen, and S. Wei, *Localizing configurations in highly-configurable systems*, in Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018, 2018, pp. 269–273.

[12] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, *Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack*, Empirical Software Engineering, (2018).

[13] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and P. Heymans, *Yo variability! JHipster: A playground for web-apps analyses*, in Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS '17, New York, NY, USA, 2017, ACM, pp. 44–51.

[14] M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. Van Horn, *Incremental computation with names*, Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (2015).

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, *Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines*, IEEE Transactions on Software Engineering, 40 (2014), pp. 650–670.

[16] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, *Configurations everywhere: Implications for testing and debugging in practice*, in Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, New York, NY, USA, 2014, ACM, pp. 215–224.

[17] G. Konat, S. Erdweg, and E. Visser, *Scalable incremental building with dynamic task dependencies*, in 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2018, pp. 76–86.

[18] C. Lamb and S. Zacchiroli, *Reproducible builds: Increasing the integrity of software supply chains*, IEEE Software, 39 (2022), pp. 62–70.

[19] C. Macho, S. McIntosh, and M. Pinzger, *Extracting Build Changes with BuildDiff*, in Proc. of the International Conference on Mining Software Repositories (MSR), 2017, p. 368–378.

[20] H. Martin, M. Acher, J. A. Pereira, L. Lesoil, J. Jézéquel, and D. E. Khelladi, *Transfer learning across variants and versions: The case of linux kernel size*, IEEE Trans. Software Eng., 48 (2022), pp. 4274–4290.

[21] G. Maudoux and K. Mens, *Correct, efficient, and tailored: The future of build systems*, IEEE Software, 35 (2018), pp. 32–37.

[22] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, *A comparison of 10 sampling algorithms for configurable systems*, in Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, 2016, pp. 643–654.

[23] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, *A quantitative analysis of variability warnings in linux*, in Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '16, ACM, 2016, pp. 3–8.

[24] N. Mitchell, *Shake before building*, ACM SIGPLAN Notices, 47 (2012), p. 55.

[25] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, *Using Others' Tests to Identify Breaking Updates*, in Proc. of the International Conference on Mining Software Repositories (MSR), 2020, p. 466–476.

[26] X. Niu, n. changhai, H. K. N. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang, *An interleaving approach to combinatorial testing and failure-inducing interaction identification*, IEEE Transactions on Software Engineering, (2018), pp. 1–1.

[27] D. Olewicki, M. Nayrolles, and B. Adams, *Towards language-independent brown build detection*, in Proceedings of the 44th International Conference on Software Engineering, ICSE '22, New York, NY, USA, 2022, Association for Computing Machinery, p. 2177–2188.

[28] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher, *Towards Incremental Build of Software Configurations*, in ICSE-NIER 2022 - 44th International Conference on Software Engineering – New Ideas and Emerging Results, Pittsburgh, PA, United States, May 2022, pp. 1–5.

[29] G. A. Randrianaina, X. Tërnava, D. E. Khelladi, and M. Acher, *On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study*, in ICSE 2022 - 44th International Conference on Software Engineering, Pittsburgh, Pennsylvania / Virtual, United States, May 2022, pp. 1–12.

[30] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, *Automated localization for unreproducible builds*, in Proceedings of the 40th International Conference on Software Engineering, ICSE '18, New York, NY, USA, 2018, Association for Computing Machinery, p. 71–81.

[31] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie, *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), 2019, pp. 527–538.

[32] Z. Ren, S. Sun, J. Xuan, X. Li, Z. Zhou, and H. Jiang, *Automated patching for unreproducible builds*, in Proceedings of the 44th International Conference on Software Engineering, ICSE '22, New York, NY, USA, 2022, Association for Computing Machinery, p. 200–211.

[33] M. Sayagh, N. Kerzazi, and B. Adams, *On cross-stack configuration errors*, in Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, 2017, pp. 255–265.

[34] R. W. Schwanke and G. E. Kaiser, *Smarter recompilation*, ACM Trans. Program. Lang. Syst., 10 (1988), p. 627–632.

[35] Y. Shi, M. Wen, F. R. Cogo, B. Chen, and Z. M. Jiang, *An experience report on producing verifiable builds for large-scale commercial systems*, IEEE Transactions on Software Engineering, 48 (2022), pp. 3361–3377.

[36] C. Song, A. Porter, and J. S. Foster, *itree: Efficiently discovering high-coverage configurations using interaction trees*, IEEE Transactions on Software Engineering, 40 (2014), pp. 251–265.

[37] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, *A classification and survey of analysis strategies for software product lines*, ACM Computing Surveys, (2014).

[38] W. F. Tichy, *Smart recompilation*, ACM Trans. Program. Lang. Syst., 8 (1986), p. 273–291.

[39] C. Yilmaz, M. B. Cohen, and A. A. Porter, *Covering arrays for efficient fault characterization in complex configuration spaces*, IEEE Transactions on Software Engineering, 32 (2006), pp. 20–34.

[40] S. Zhang and M. D. Ernst, *Automated diagnosis of software configuration errors*, in 2013 35th International Conference on Software Engineering (ICSE), May 2013, pp. 312–321.