PyroBuildS: Enabling Efficient Exploration of Linux Configuration Space with Incremental Build

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

PyroBuildS: Enabling Efficient Exploration of Linux Configuration Space with Incremental Build

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

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

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

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

ABSTRACT

Software engineers are acutely aware that the build of software is an essential but resource-intensive step in any software development process. This is especially true when building large systems or highly configurable systems whose vast number of configuration options results in a space explosion in the number of versions that should ideally be built and evaluated.

Linux is precisely one such large and highly configurable system with thousands of options that can be combined. Previous study showed the benefit of incremental build, however, only on small-sized configurable software systems, unlike Linux. Although incremental compilation for post-commit is used in Linux, we show in this paper that the build of large numbers of random Linux configurations does not benefit from incremental build.

Thus, we introduce and detail PyroBuildS, our new approach to efficiently explore, with incremental builds, the very large configuration space of Linux. Very much like fireworks, PyroBuildS starts from several base configurations ("rockets") and generates mutated configurations ("sparks") derived from each of the base ones. This enables exploring the configuration space with an efficient incremental build of the mutants, while keeping a good amount of diversity. We show on a total of 2520 builds that our PyroBuildS approach does trigger synergies with the caching capabilities of Make, hence significantly decreasing builds time with gains up to 85%, while having a diversity of 33% of options and 15 out of 17 subsystems. Overall, individual contributors and continuous integration services can leverage PyroBuildS to efficiently augment their configuration builds, or reduce the cost of building numerous configurations.

1 INTRODUCTION

Building software is a crucial activity for developers and maintainers of projects. Various artefacts are assembled, compiled, tested, and then deployed, presumably successfully. The emergence of continuous integration (CI) has accelerated this trend with the integration of build services into major code platforms (*e.g.*, GitHub, GitLab). The goal is to continuously ensure some quality assurance of software products, whether in terms of functionality or non-functional properties (*e.g.*, security, execution time).

Software configurations add further complexity to the problem of building software. Different variants of the artefacts can be assembled *e.g.*, due to conditional compilation directives `#ifdef`-s in the source code. Different external libraries can be compiled and integrated as well. The way the build is realized can also change *e.g.*, with the use of different compiler flags. Developers and maintainers of a project want to ensure that, throughout the evolution, all or at least a subset of software configurations build well. As most of today's software is configurable in order to fit constraints, functional and performance requirements of users, it is not surprising to observe that many organizations build different software configurations of their projects. For instance, initiatives like KernelCI or 0-day build thousands of default or random Linux configurations each day [2, 3, 48, 63]. Another example is JHipster, a popular Web generator, that builds dozens of configurations at each commit, involving different technologies (Docker, Maven, grunt, etc.) [25].

Highly configurable systems like the Linux Kernel need intensive testing –especially for Linux which is used in critical systems. Linux 6.1 has over 20,000 configuration options, each of which can be either enabled, enabled as module or disabled by the user, thus shaping the generated executables. Considering that each of these 20,000 options can take only two values, enabled or disabled, leads to 2^{20000} possible configurations. The actual number is lower, due to constraints among features, but remains huge. It is thus practically infeasible to test all the possible configurations. However, part of the configuration space can be tested through uniform random sampling of the configuration space. Getting a uniform random sampling of such a huge configuration space still is an open question [28, 47], but the Linux kernel developer have written a widely used tool to generate random configurations.

Kernel builds are important for the Linux community for verifying that kernels compile well regarding different architectures and configuration options – builds are also a prerequisite to check whether kernels boot, pass test suite. Hence, multiple initiatives exist that intensively build Linux kernel configurations, such as KernelCI [3], Intel 0-day [2, 36], Tuxmake [59, 63] or TuxML for instance [5, 47]. However, building software is increasingly complex and costly in terms of time and resources, such as energy [11, 17, 29, 43]. For instance, the TuxML project took more than 15K hours of computation time to build 90K+ random configurations, and derive an accurate prediction model for size prediction [47].

KernelCI builds about 400 configurations per-day, with reports on the Linux Kernel Mailing List (LKML) when errors occur. KernelCI mostly tests default configurations that must build and also considers additional, random configurations if the computational resources allow it. The kernel needs to test more *esoteric* configurations to further explore the configuration space in diverse settings and ensure quality assurance throughout the evolution.

In this paper, we thus aim at tackling the issue of speeding up builds of configurable software: It can help to diversify and augment the number of tested configurations, or it can help to reduce the cost (*e.g.*, computational time) of a build campaign. We propose PyroBuildS, an automatic approach relying on incremental builds of configurations. The principle is to reuse the result of an existing build for other configurations. In contrast to traditional build that would clean and restart from scratch, some artefacts do not have to be re-built. The key ingredient of PyroBuildS is to select configurations that are close to each other – otherwise, the distance between configurations is too important and compiled artefacts cannot be shared across builds. Hence, PyroBuildS synthesizes numerous configurations out of a configuration base through an innovative mutation mechanism. Very much like fireworks, and to give an image, we produce many sparks (hence the term "pyro") around a base configuration that allow for diversification of options, without deviating too much to keep the incremental build efficient.

We target the Linux configurable software as it is highly configurable and complex software. In [56, 57], Randrianaina *et al.* explored the idea of incremental build of software configurations, but on much smaller configurable software with fewer options and possible configurations. Furthermore, empirical results show some limitations (*e.g.*, incremental build of configurations is not always correct) and potential benefits (*e.g.*, the reduction of build time can be achieved under the conditions that a specific ordering is found). The exploratory study raised the open challenge of finding a strategy that is both correct and effective, especially at the scale of the Linux kernel configuration space. Thus, we first replicate their hypothesis on a Linux, which is far larger than subject systems considered in [56, 57] *w.r.t.* number of options (15K+ options vs less than one hundred), configurations, and lines of code. We empirically show that there is no benefit for incremental build with randomly selected Linux configurations – worse, the computational time is much more important. We then used PyroBuildS to incrementally build Linux configurations with a succession of mutations. This later case showed significant benefits in comparison to a traditional clean build. On a total of 2520 builds, we observed a systematic gain going up to 85%, while reaching on average 90.6% correctness, 33% diversity of options and 15 out of 17 subsystems that a random Linux configuration would explore in terms of activated options and subsystems.

Overall, our contributions are as follows:

- PyroBuildS's approach that offers (1) automated mutation of configurations; (2) progressive exploration of configuration space with incremental builds.
- Empirical results that show the inefficiency of incremental build between random configurations of Linux;
- An assessment of PyroBuildS showing the significant gain when a base Linux configuration (*e.g.*, default configuration or random configuration) is mutated and incrementally built. PyroBuildS also provides a good diversity, while resulting incremental builds are strictly similar to traditional builds in a vast majority of cases.

This paper is organised as follows. Section 2 provides background material on Make, incremental build, and on the Linux kernel build system. Section 3 outlines the motivation for our work, and Section 4 explains our approach for PyroBuildS. Section 5 presents the experimental setting, evaluation and results of our work. Section 6 discusses our work, results and threats to validity. Related work in analysed in Section 7. Section 8 concludes and outlines future work directions.

2 BACKGROUND

This section provides some background on the Make build system and incremental build, then on the Linux kernel build system.

2.1 Make and Incremental Build

A build system specifies how to translate source code into a deliverable. The developer declares rules on how to compile a source code, which libraries to include and how to link them all together in order to obtain an executable. Make [19] is one of the most popular build systems [52].

Make. Build specifications are declared in *Makefile*s that are spread out in the subdirectories of a project. Makefiles contain rules that are either a filename to generate a library or object/executable file, with its "recipe" depending on source files; or rules that are not filename-based but rather rules to trigger other rules. *Make* starts by reading the *Makefile* at the root of the project then recursively goes into subdirectories. On its way, *Make* builds a dependency graph of the relations between rules and spawns a build process to satisfy each rule.

On its first invocation, *Make* checks if rules are satisfied. If the target file does not exist, *Make* produces it according to the specification. Otherwise, it checks the timestamps of its dependencies: if the target file is older than one of its dependencies, it must be recompiled. This process is propagated on the whole dependency graph. For a first build of a project, *Make* does a full build because none of the target files exist yet.

Incremental build. After a full rebuild all rules are satisfied. A second invocation of *Make* does nothing if no modification was done. However, if a modification is done on a source file, a call to *Make* checks timestamps and updates the rules impacted by the last modification. This process does not rebuild the whole project every time, if the specifications are correctly written[13], thus saving resources. This process of rebuilding only the minimum necessary parts of the build is called an *Incremental build*.

2.2 The Linux Kernel Build System

The Linux kernel build system is based on *Make*, and is separated in two parts: *Kconfig* which lets the user configure the system and *Kbuild* that translates this configuration into build specification and runs the build process to generate the deliverable.

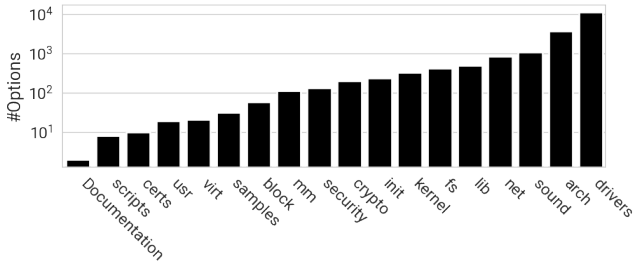


Figure 1: Configuration options per subsystem in Linux 5.13 (log scale).

Kconfig. Configuration options are specified in `Kconfig` files in the kernel source tree. They contain the name of the option, its type (*boolean/tristate/hex/string/int*), its dependencies, a help description if necessary or an import of other Kconfig files. Usually, one or more Kconfig file(s) describe options implemented in a directory and imports the Kconfig files of its subdirectory recursively. Hence, the Kconfig file at the top level of the source tree just contains a set of imports to the other subdirectories of Linux. Figure 1 shows the repatition of configuration options inthe Linux Kernel.

Kconfig assists the user in the configuration of the kernel via command line or with a GUI. It has also a set of default configurations such as *tinyconfig* (few options, small binary size, suited for embedded systems), *defconfig* (default configuration for a given architecture), *randconfig* (random option choices) and *allyesconfig* (a maximum of enabled options) among others. This configuration phase produces a configuration `.config` that can be built.

Kbuild. Make is used by the Linux kernel to produce the deliverable. Linux subdirectories contain `Makefiles` that set variables such as compiler flags or version, or environment variables that set the architecture. They also specify build rules with conditionals depending on the options specified in the configuration file. In addition, *Kbuild* generates a file for each of the enabled features of the configuration file. Then, it has the rules impacted by the enabled option depend on this newly generated file. Hence, when the user modifies a specific option, only the rules that depend on this option are re-executed. This is intended to have incremental build actionable for the kernel build. In fact, most projects that use Make keep their configuration in one file (usually named `autoconf.h` or `config.h`) and all the rules of the project depend on this file. When the user modifies the value of an option for instance, the whole project is built every time since this dependency is newer than the other rules.

3 MOTIVATION

As explained in section 1, building a large number of varied configurations of the Linux kernel is an important but expensive task. Although a number of works and techniques exist to optimize the compilation (*i.e.*, build) of systems *after commits* [1, 14, 18, 20, 27, 39, 50, 54, 60, 65], resulting in deployments such as the one of `ccache`[1], a much lower number of works pertain to optimising (ie. decreasing the cost of) massive builds of Linux kernel configurations. The most commonly deployed solution is quite costly, since it consists in parallelizing the builds, thus saving wall time but not decreasing resources usage at all.

The current practice when building Linux kernels is to perform clean builds (see section 2.2). Usually, CI infrastructures build default configurations as mentioned in Section 2.2 for various architectures of the kernel, and for all patches or new releases of Linux these default configurations must at least build. Some previous works such as [56] have made first inroads in the direction of incremental builds for Linux. However, although the authors reported good results for small `x264` systems, their attempts with incremental builds of the much larger Linux kernel did not bring any significant decrease in resource consumption.

In the work we report in current paper, we tried to explore and understand how and under which conditions could incremental builds be used with benefits for the Linux kernel too.

Our initial intuition is better explained with an example. Typically, when configuring the Linux kernel, a user or a developer starts with a base configuration and applies some changes until the kernel matches the required usage. The default configuration on the considered hardware architecture — `x86_64` for example — can be taken as a starting point. First, this base configuration is built — which takes 92s in our example. Then, to find the configuration that satisfies a specific need, options can be disabled or enabled; for instance we disable support for Network FileSystems, enable Hardware Monitoring Chip debugging messages. Then we build the modified configuration. However, instead of performing a clean build, the previous build of the default configuration can be reused — thus instead of waiting for an extra 92s again to build the new configuration, the incremental build took 7s only

Our approach is thus based on this observation. We seek in which extent such configuration mutations can be beneficial to incremental explore the configuration space at lower cost, while retaining a significant diversity of the built configurations.

In the next section, we thus present our PyroBuildS approach, that progressively mutates configurations in order to inject diversity while relying on incremental build to rebuild only parts that are necessarily added by the last mutation.

4 APPROACH

This section first discusses the naive approach of incremental build, before introducing our approach of PyroBuildS with mutation-based incremental build and its implementation.

4.1 Naive incremental approaches don't work

Based on the observations of the previous Section, a first naive approach can be to keep diversity by successively building random configurations, while trying at the same time to reap the fruits of incremental builds by not doing any `make clean` between the builds. This way each build could benefit from the caching of the artefacts created by the previous build(s).

Figure 2 shows the results of incremental builds of 50 random configurations using Make.

[1] https://ccache.dev/

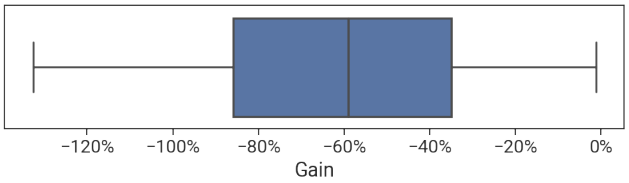


Figure 2: Overhead of incremental build on random configurations.

We can observe that incremental build does not bring any time benefit; on the contrary, incremental build can even double build time of some configurations.

This can be explained by the immensity of the configuration space of Linux, as explained in Section 1, and by the fact that random configurations built are very diverse, with up to more than a thousand of option differences. Hence, configurations that are completely different can be obtained, which implies completely rebuilding the new configuration. During an incremental build, Make checks whether some dependencies have already been built, but because of the huge difference between configurations, no rule can be reused. In such a case, a full (re)build is necessary, so no time is saved. Even worse, all the checks performed by Make on the already build artefacts (from previous configurations) are additional overhead.

This first experiment shows that just adding plain, naive incremental building is not producing any synergies, because of the sheer size of Linux configuration space.

It is thus necessary to actively increase the locality of successive incremental builds to actually reap benefits. This is precisely what our new PyroBuildS approach and tool are designed for.

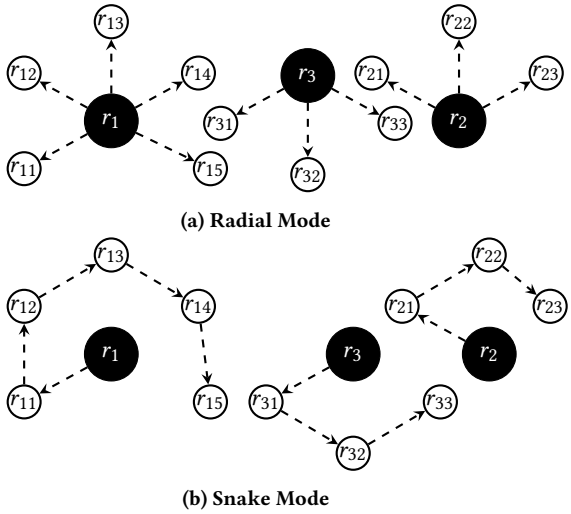
4.2 PyroBuildS overview

Aiming at taking full advantage of incremental builds when building the Linux kernel, our PyroBuildS features exploration strategies designed to harness Make's incremental build capabilities [2]. This is done by relying on the locality, in the configuration space, of the explored build configurations, to maximize the effectiveness of the technical mechanisms that compose the build system. The rationale is that the locality of the mutations in the configuration space increases the probability of reusing previously built artifacts, hence decreasing average and thus amortised build time. A proper amount of diversity must however be kept, because one of the *raisons d'être* of building various configurations is to verify that they do build, hence that the code is correct.

PyroBuildS thus integrates a configuration mutator, whose strategies will guide the way it explores the configuration space. Our mutator is less aggressive than a full configuration change like *randconfig*, since it progressively mutates a configuration, thus benefiting from Make's incremental build caching.

PyroBuildS first builds an initial valid configuration, called *base configuration*. Then it randomly picks an option and computes its options dependencies. The resulting mutation that is actually applied to the configuration is the addition of the chosen option mutation and all the implied mutation on its option dependencies. The resulting mutated configuration is then built. This build is an incremental one, using the build directory of the base configuration. Applying this process multiple times according to some exploration strategy varies the configurations and progressively explores the configuration space.

[2]Other forms of caching, e.g. Ccache, can also benefit from the synergies created by our PyroBuildS incremental build approach, but due to space constraints these are outside the scope of this paper.



(a) Radial Mode

(b) Snake Mode

Figure 3: Exploration strategies.

PyroBuildS exploration can follow two strategies, *radial exploration* and *snake exploration*, depicted in Figure 3. The *radial exploration* always mutates the base configuration. Its goal is to explore the configuration space with slight modifications from a base (see Figure 3a). The *snake exploration* starts by mutating the base configuration, then after each build, mutates the last mutant. It thus accumulates the mutations and the configurations built through iterations (see Figure 3b).

Both *radial exploration* and *snake exploration* can be seen as fireworks, where a rocket (base configuration) is shot, and explodes in sparks (mutations) that are either spread radially around the rocket (*radial exploration*) or create a snake-like trail of sparks (*snake exploration*). Hence the name PyroBuildS, based on the Greek word *pyr* meaning fire.

4.3 PyroBuildS implementation

The base configuration is usually one of the default configurations of the Linux kernel: *tinyconfig*, *defconfig*, *allyesconfig*. It is also possible to start with a *randconfig* with options' values set randomly. The *randconfig* tool is widely used in the Linux community to test the kernel in different settings [4, 5, 47, 53].

PyroBuildS's implementation is depicted in Figure 4. First, an option is randomly picked over all Kconfig options supported by the chosen architecture ①. PyroBuildS mutates over *boolean*s and *tristate*s only, randomly picking a value for the chosen options ②: either *yes* or *no* for a boolean, and *yes* or *no* or *mod* for a tristate. If the chosen option is already set in the current configuration, a value different from its current one is chosen. The configuration and

mutation are then fed to the mutator that generates the resulting mutated configuration.

Note that a modification on a configuration option can impact other options, so dependencies and possible conflicts with the newly added option have to be solved. This is done using ConfigFix [22] ③, which relies on a SAT formula to represent the constraints of Kconfig, and finds a logically valid solution (*i.e.*, a new configuration) for a mutation over the current configuration. When the picked option cannot be not solved by ConfigFix [22], we iterate over the mutator until finding an option that is solved by ConfigFix. Eventually, the mutation is applied with its dependencies and a new configuration is generated ④.

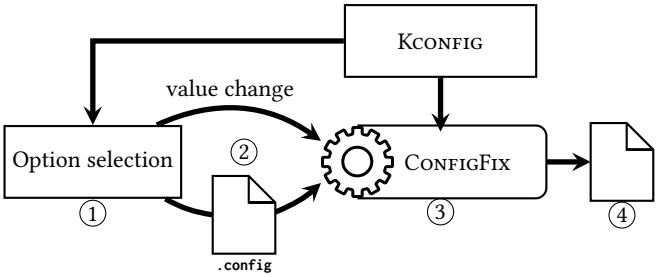


Figure 4: PyroBuildS mutation procedure.

5 EVALUATION

In this section, we first present and detail the insights of our research questions, then we explain our experimental settings.

5.1 Research questions

RQ_1 *(Correctness and consistency) Are PyroBuildS incremental builds correct and consistent with clean builds?*
We must ensure that the output of an incremental build is the same as the output of a clean build. We verify *correctness* by comparing the name and size of each symbol in the binary using the `bloat-o-meter` Linux kernel tool. Since the kernel introduces metadata like build date or build version in the binary, a bit-by-bit comparison always fails. We consider the build *correct* if `bloat-o-meter` does not detect any changes between two binaries of the same configuration with clean and incremental build.

RQ_2 *(Cost reduction) Are PyroBuildS incremental builds faster than clean builds?*
Our main goal is to accelerate the build of configurations with incremental builds. Here, we measure the build time of an incremental build, then we compare it to the clean build time.

RQ_3 *Does PyroBuildS diversify configuration space exploration?*
The mutation of configurations is intended to be beneficial for the build property, which is build time coupled with incremental build. However, the main goal of CI infrastructures is to test many configurations to cover the configuration space as much as possible. This research question evaluates the mutations of PyroBuildS in terms of option diversification.

5.2 Experimental settings

We conducted three major experiments. First, we evaluate the exploration strategies described in Section 4 using only random configurations as configurations, i.e., without any mutation. Second, we evaluate the PyroBuildS approach on both its configuration mutation and its impact on incremental build with two kinds of bases, randconfig and defconfig, that we mutate in radial and snake modes.

When we generate a random configuration, we preset some values in the configuration. The architecture (Intel x86 and 64 bits) is predefined, and we disable a compiler option `GCC_PLUGIN_CYC_COMPLEXITY` that is a demonstrative option to show how to write a GCC plugin and does not have any effect on the binary[3].

The experiment starts with a base configuration that we clean build. Then we use our mutation operator to mutate this configuration to get a new mutant. For our experiment using randconfigs only, this mutation phase is replaced by a generation of a new randconfig. In a *radial exploration*, we incrementally build the new configuration from a directory on which a clean build of the base has been performed. For a *snake exploration*, we chain up the builds such that each incremental build is performed in the directory in which the build of the previous configuration had been performed.

In order to track some differences and keep the intermediate files and final binaries of our experiment, we store everything inside a local git repository. We initialize this local git repository from the source code of the Linux kernel which represents the main branch. Then, we derive a new branch from it, in which the configuration to build is copied. Finally, we build the configuration and force the staging of all our modifications and commit it in order to do a snapshot of the experiment. Hence, a clean build is a build made in a new branch that has the main branch – which is the clean source code– as parent and an incremental build is a build made in a new branch that has another build branch as parent.

In order to answer our research questions, we also perform a clean build of all of the configurations we previously built incrementally to compare some metrics.

For the experiments, we start by generating a fixed set of 20 configurations. Then, for each exploration strategy we mutate each one of these 20 bases to 10 mutants that are built incrementally according to the chosen strategy. For one exploration strategy, we do 20 clean builds of the base, then 20×10 incremental builds. In addition, we do 20×10 clean builds for each of the explored configurations to compare them afterward. Since we evaluate 2 exploration strategy in 3 ways, we perform 2520 builds in total.

Using the presented build approach, we conduct the experiment with the Linux kernel as a subject system.

We run our experiments on two servers with respectively 2xAMD EPYC 7H12 64-Core, 503 Go of RAM and 2xAMD Epyc 7532 CPU and 512 Go of RAM. Our experiments run inside a container from a Docker image based on Tuxmake's[4] on which we added some extra tools for our measurements.

5.3 Results

We now answer our research questions.

[3]https://cateee.net/lkddb/web-lkddb/GCC_PLUGIN_CYC_COMPLEXITY.html
[4]https://hub.docker.com/r/tuxmake/x86_64_gcc-11/

5.3.1 RQ_1: (Correctness and consistency) Are PyroBuildS incremental builds correct and consistent with clean builds? We compare the binaries obtained by a clean build and by an incremental build of the same configuration. Building the Linux kernel in order to get reproducible binaries is not trivial if we refer to the official documentation[5]. Indeed, metadata such as the date of the build, the host username and others are embedded in the binary. These kinds of metadata render the build not reproducible and bit by bit comparison too restrictive. Though we set some environment variables related to the build system during our builds, namely `KBUILD_BUILD_TIMESTAMP`, `KBUILD_BUILD_USER`, `KBUILD_BUILD_HOST`, `KBUILD_BUILD_VERSION`, we do not handle modules signing keys and structure randomization.

We base our comparison criteria on the symbols contained in binaries. First, the binaries of the same configuration are compared, then their symbols and sizes are compared. To do that, we use the Linux kernel utility `bloat-o-meter`[6]. We consider the binary from an incremental build to be *correct* if `bloat-o-meter` does not detect any differences with the binary from a clean build.

We also consider, as the *consistency* criterion, that when the clean build fails, the incremental build should fail too. Otherwise, the incremental build is considered inconsistent.

Table 1 shows the correctness percentages of incremental builds compared to their respective clean builds. With defconfig, incremental builds always provide correct binaries. However, some binaries that are not correct are obtained when using random configurations as base. First, with PyroBuildS' mutations, the binaries obtained in the *radial exploration* are correct at 80%. For the *snake exploration*, binaries are 94.5% correct. With the exploration where, instead of applying PyroBuildS' mutations, new randconfigs are generated, the correctness rate is around 85%.

We found out that this was due to the value of a string that depends on the state of the source git repository. In fact, the content of string `VERMAGIC_STRING` is marked as "-dirty" if the repository has some unstaged changes. Since the repository has some unstaged changes right after a build, the product of an incremental build has this additional tag in this string. This issue is documented in the documentation on Linux reproducible build mentioned earlier.

Regarding consistency, PyroBuildS always ends with consistent incremental builds. However, the case when only random configurations are chained results in builds that fail in incremental build but do not fail in clean build. This happens in 97% of the cases for the full-random method with snake and 98% for radial.

> ***RQ_1* insights:** Incremental build using progressive mutations of PyroBuildS are *correct* at 100% with a defconfig base, 80% with a random configuration as base for radial and 94.5% for snake. All builds are *consistent* compared to full randconfigs.

5.3.2 RQ_2: Are incremental builds fast? To answer this research question, we first experiment incremental build with random configurations, then with random mutations to the base configuration.

[5]https://www.kernel.org/doc/html/latest/kbuild/reproducible-builds.html
[6]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/bloat-o-meter?h=v5.13

Table 1: Correctness and consistency.

Base	Mode	Correctness (%)	Consistency (%)
Default	radial	100%	100%
	snake	100%	100%
Random	radial	80.5%	100%
	snake	94.5%	100%
	snake (Random)	85.2%	97.0%
	radial (Random)	85.4%	98%

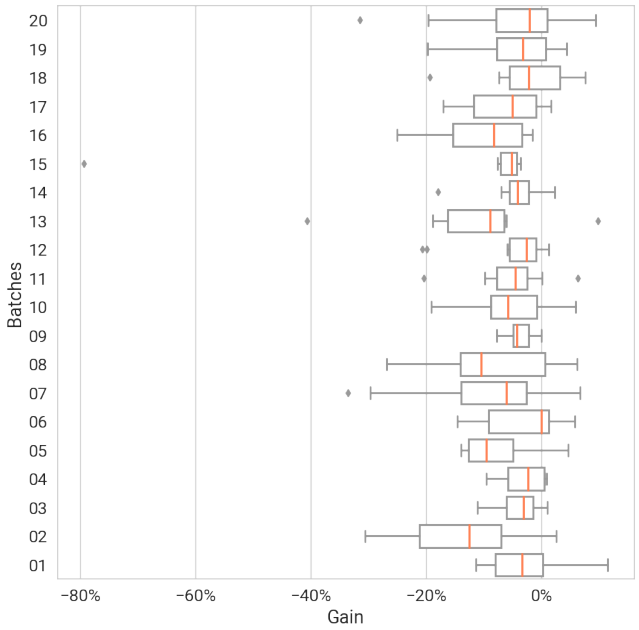


Figure 5: Build time gain with full randconfigs in radial exploration.

Figures 5 and 6 depict the build time box plot for the incremental build of random configurations. We observe on average a systematic loss for incremental build time compared to the clean build of the random configurations, both in radial and snake strategies. The loss can go up to −79% for radial and −288% for snake exploration.

Figures 7 and Figure 8 show the time performance gains of the incremental build with random mutations from the default configuration `x86_64` as a base. We observe on average a systematic overall gain for incremental build time compared to the clean build of random configurations, both in radial and snake strategies. The gain can go up to beyond 80% for both cases, with loss going down to −20% for the snake exploration especially.

Figures 9 and Figures 10 show the time gains of incremental build with random mutations from a random configuration as a base. We also observe on average a systematic overall gain for incremental build time compared to the clean build of the random configurations, both in radial and snake explorations. In both case, the loss can be near −20% while the gain can go above 80%.

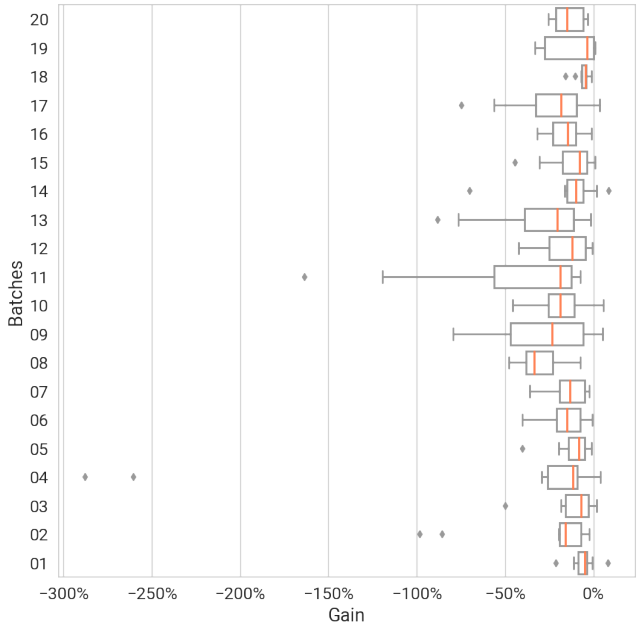


Figure 6: Build time gain with full randconfigs in snake exploration.

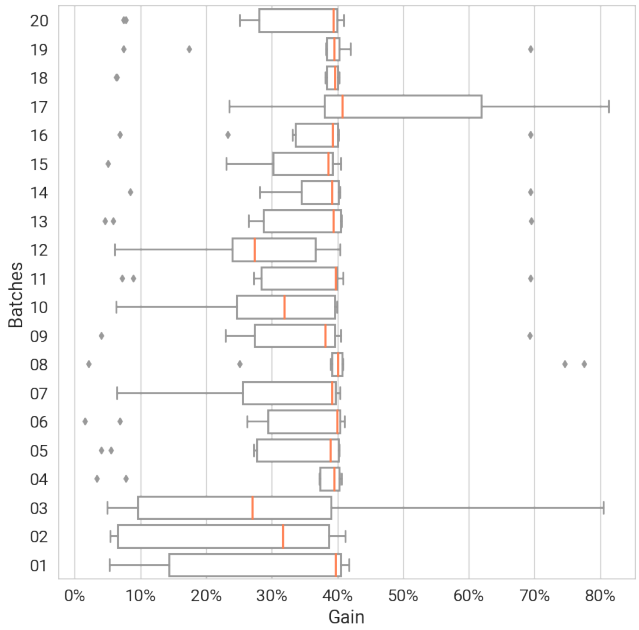


Figure 7: Build time gain for PyroBuildS with radial exploration with x86_64 default configuration as base.

While the gains are observed for both the default configuration x86_64 and a random configuration when used as a base, the gains are more prominent and significant in the former.

Finally, Figure 11 aggregates overall time performance when incremental build is applied with mutations.

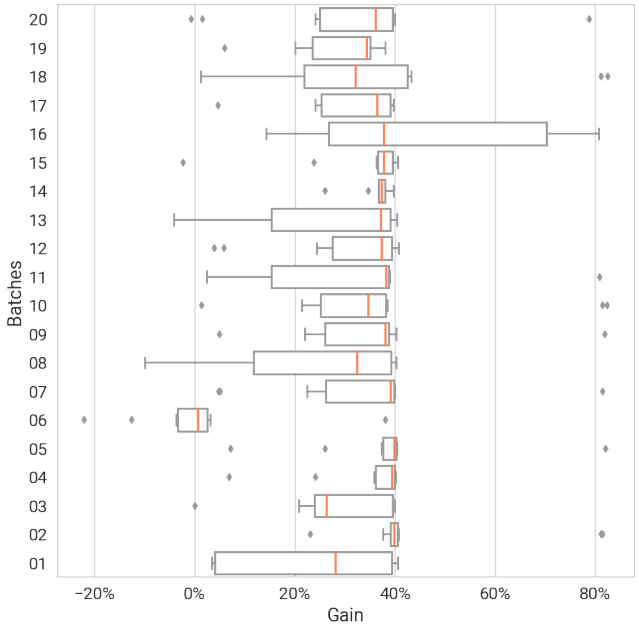


Figure 8: Build time gain for PyroBuildS with snake exploration with x86_64 default configuration as base.

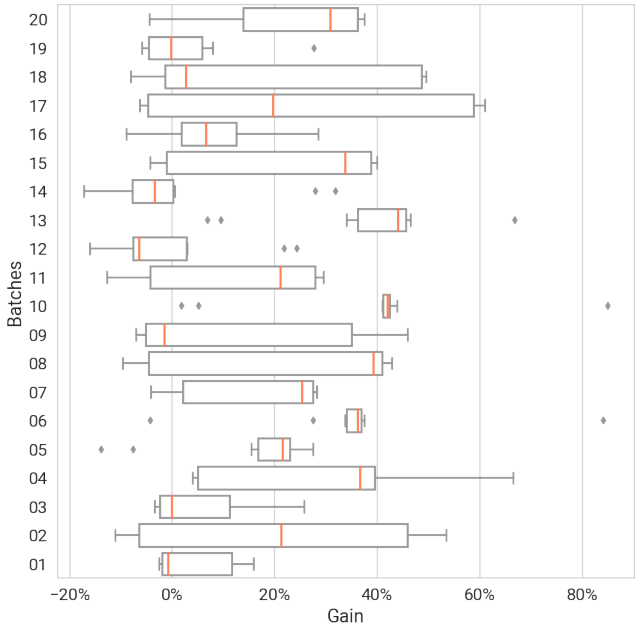


Figure 9: Build time gain for PyroBuildS with radial exploration with random configurations as base.

***RQ*$_2$ insights:** With random configurations in radial and snake explorations, incremental build looses respectively up to −79% and −288%. However, with PyroBuildS mutations, both with random and default configurations, the loss is contained to less and only −20% with a gain up to and beyond 80%.

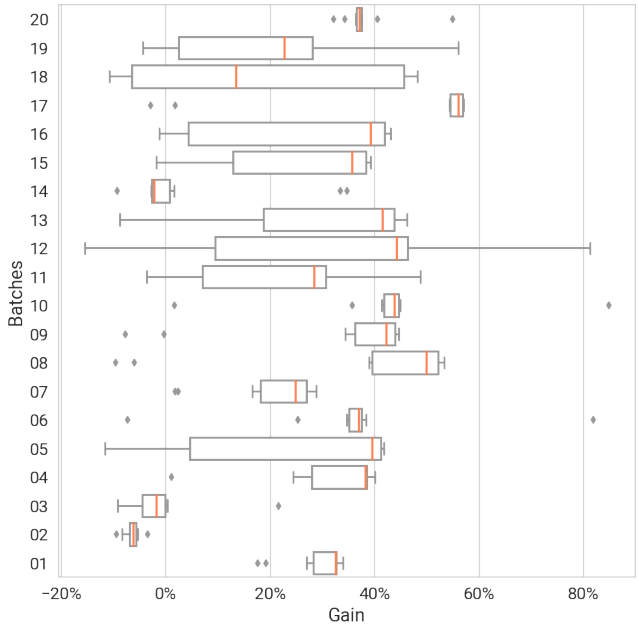


Figure 10: Build time gain for PyroBuildS with snake exploration with random configurations as base.

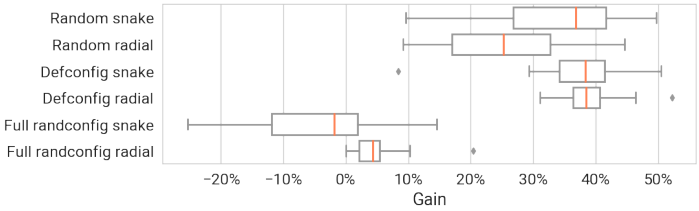


Figure 11: Total gain snake exploration vs radial exploration.

5.3.3 RQ_3: Does PyroBuildS diversify configuration space exploration? To answer the questions centered on the diversity of build configuration, we compare the diversity produced by PyroBuildS to the one produced by random configurations. In particular, we look at the number of options that are impacted by our changes and the number of targeted subsystems.

In Figure 12, we present the number of all options impacted by our changes in the presented explorations. An impact over an option is anything that can have an effect on its value: value change (*e.g.*, from yes to module), disabling or enabling it.

As baseline, we consider the diversity of randconfigs. In fact, randconfigs chose options randomly, thus having configurations spread out in the configuration space. Even though there is no particular gain in time, randconfigs cover well the configuration space.

For each mutant of PyroBuildS, whatever the exploration strategy, we compare them with the base by keeping the options that were modified. Then, we check in which subsystem they occur. We also include the base configuration in our data since it serves to explore the configuration too.

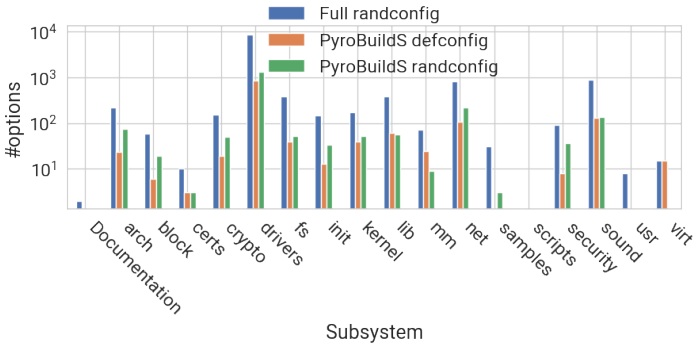


Figure 12: Diversity of options (log scale) per subsyetm.

From Figure 12, we observe that PyroBuildS is able to reach 15 out of 17 subsystems that are covered by randconfigs. Moreover, PyroBuildS was also able to cover 33% of the options that are covered by randconfigs. Thus, the idea of combining random configurations and mutation ("sparks") as part of PyroBuildS is validated, offering local and global diversity for exploring the configuration space.

> ***RQ_3* insights:** Diversity of the PyroBuildS covers 15 out of 17 subsystems and 33% of the options that both are covered by randconfigs. Overall, mutation-based builds (1) provide a tradeoff between diversity, build time, and correctness; (2) are an interesting complement to random configurations.

6 DISCUSSION AND THREATS TO VALIDITY

We now discuss the impact of our approach and observed results, then the threats to validity.

6.1 Impact and Open Directions

Recommendations for practitioners. Owing to our empirical results, we can synthesize the following actionable recommendations:

- Random configurations of Linux should not be built incrementally – due to the huge configuration space, the distance between configurations is simply too important, and can even lead to a (significant) increase in time;
- Contributors, testers, and janitors of the Linux kernel can use PyroBuildS when using the defconfig configuration, which is widely considered for testing the non-regression of patches. Once it has been built and tested, an opportunity exists to build numerous mutated configurations out of defconfig, hence diversifying the number of tested configurations at a lower computational cost. Continuous integration (CI) initiatives like KernelCI or 0-day can also leverage PyroBuildS to diversify their set of tested configurations once defconfig has been built;
- CI services in the Linux ecosystem can leverage PyroBuildS to further explore and test the configuration space. The recommended usage is to select some random configurations, and then systematically mutates them with PyroBuildS.

The different mutations ("sparks") can be done independently and parallelized using different machines.

Further applications. Building kernel configurations is a key activity in Linux development, mainly for quality assurance. Throughout the configuration space exploration, a possible application is to report failures (and hopefully fix configuration-related bugs). As future work, we plan to investigate whether configuration diversity (as synthesized by PyroBuildS and random configurations) correlates to bug-finding ability. Though we have not designed PyroBuildS specifically for finding configuration-related failures, it is possible to control the exploration strategy through mutation. Hence, we envision to prioritize some options prone to failures or considered as more important *w.r.t.* testing. A possible usage of PyroBuildS is to incrementally build mutated configurations out of a build that already fails, aiming at isolating the combination of options that cause the failure. In this paper, we focus on the main case of mutating over a correct configuration build.

Another possible application is to leverage PyroBuildS to synthesize training sets for prediction models of configurable systems. Specifically, prior works show that machine learning models can predict quantitative properties of Linux configurations, but these models require lots of observations and measurements over configuration builds (*e.g.*, 10K+ random configuration builds [47]). Hence, PyroBuildS can provide a way to obtain larger training sets (with more configurations) with the same computational cost (*e.g.*, time), or to obtain training sets with the same number of configurations but with less computational cost. In particular, PyroBuildS can be seen as a way to augment data in a cost-effective way. An open question is to what extent would the diversity of mutated configurations benefit to learning methods.

Further possible improvements. PyroBuildS could be extended with a compiler cache like Ccache, in order to further increase the reusability of already build artefacts. This is left for future work, as we primarily focused on the effect of incremental build itself. Another research direction is to construct a *deny list* that would prevent some configuration options from being mutated as part of PyroBuildS. Intuitively, some options are cross-cutting many artefacts and changing their values boils down to recompiling everything. Hence, PyroBuildS should avoid such ineffective situations. Two challenges arise, however. The first is to find a comprehensive and effective deny list, either with domain knowledge, static analysis, or patterns extracted out of PyroBuildS observations. A second challenge is to retain diversity. As an extreme case, all options could be part of the deny list except independent options (mostly drivers); this would likely be effective but yield a very low diversity.

Comparison with prior studies. We now discuss some of the results of Randrianaina *et al.* [56, 57] compared to our observations and insights. First, in terms of correctness (RQ1), Randrianaina *et al.* report that 57.80% of incremental builds give the same binaries. In the case of Linux, we obtained much better correctness results. An explanation is that the Linux build system (with Kbuild) is aware of configurations specifics and is effective to support incremental build. It is not necessarily the case for other subjects considered in [57]. In addition, we use a stronger procedure to compare binaries in PyroBuildS: instead of looking at binary size and symbol tables as in their original studies, we use the Linux integrated tool `bloat-o-meter` that is stricter. In terms of effectiveness and cost reduction (RQ2), Randrianaina *et al.* showed that the reduction of build time can be achieved under the condition that a specific *a posteriori* ordering of configurations builds is found. We did not follow this approach for two reasons. First, finding *a priori* such an optimal order remains an open challenge. Second, random Linux configurations seem simply too distant to pay off, whatever the ordering.

6.2 Threats to validity

We now discuss internal and external threats to validity [67].

Internal validity. We first measured the time performance of clean and incremental builds. The main risk lies in the interference of measurements with other running software. To counter this risk, we used a dedicated server where only our experiment was executed. We further isolated the build environment docker image with only the required build tools and dependencies. Moreover, we only ran our experiment once due to the high cost and execution time. Indeed, rather than reducing the experiment to randconfig or defconfig only, and repeat the experiment twice or more, we chose to rather diversify the data set with both defconfig and various randconfig configurations, with and without the mutations. Therefore, we mitigate the risk of over-fitting the overall observed results in Figure 11 by diversifying our data set and built configurations. Nonetheless, before the experiment, we used the HyperFine tool[7] to benchmark the machine with a defconfig. Hyperfine showed that the variance of the defconfig build is lower or equal to 3%, which gives what we consider an acceptable standard deviation in our experimentation.

Moreover, we reflected on diversity by looking at the number of options and subsystems that are activated during the build. Other metrics could also be considered, such as the impacted source code. However, the options and subsystems are good indicators for the diversity of a given configuration.

External validity. We experimented on the Linux kernel as a subject, which is a C-based highly complex and highly configurable software system with the Make build system. We purposely focused on it in order to replicate the previous work of Randrianaina *et al.* [56, 57] on a different scale of complex configurable software system, namely Linux. Further results of PyroBuildS with mutation-based incremental build on Linux cannot be generalized to other small-sized configurable software system, more experimentation remaining necessary in this area. Although we think that incremental build with PyroBuildS should be applicable in other build systems and software technologies, further experiments are necessary before generalizing the observed results.

7 RELATED WORKS

This section discusses the related work about build systems and software variability (configurations). To the best of our knowledge, the incremental build of configurations has caught little attention, especially at the scale of the Linux kernel.

Build systems. Many works exist on (incremental) build systems [1, 14, 18, 20, 27, 39, 50, 54, 60, 65] but they focus on code changes through the evolution of software (e.g. commits) rather

[7] https://github.com/sharkdp/hyperfine

than configurations. When considering configurations, the differences can be very important (i.e., much more important than across commits), spanning numerous files, and thus challenging the effectiveness of incremental compilation and build in this context. Cserep *et al.* [16] introduce how to detect only the necessary files to build with incremental parsing of the codebase. In [39, 40], Konat *et al.* provide a DSL to increase the effectiveness of writing build scripts. With such an expressive language, analysis and error detection could be performed beforehand. Maudoux *et al.* [49] show that incremental build could help speed up builds of continuous integration (CI). Gallaba et al. [23] proposed to accelerate the continuous integration build. They infer data from which build acceleration decisions can be made by caching the build environment and skipping unaffected build steps. An open issue is to adapt these techniques over distant software configurations that may have very different impacts on the files to build.

Several empirical studies on build systems have been performed (*e.g.*, [30, 31, 41, 44, 50, 51, 68]). Beller *et al.* [12] performed an analysis of builds with Travis CI on top of GitHub. About 10% of builds show different behaviour when different environments are used. In our case, we are considering different software configurations rather than environments. Lawall et al. [42] proposed JMake, a mutation-based tool for signaling changed lines that are not subjected to the compiler. They aim to find an appropriate Linux configuration that can compile and thus covers the files in which changes have been made. In contrast, we use mutation to diversify an existing base configuration and synthesize several related configurations that can then be incrementally build.

Software product line (SPL) and variability. The SPL community develops numerous methods and techniques to manage a family of variants (or products). Configurations are used to build or execute variants and are subject to intensive research. For instance, building variants is a necessary step before deriving performance prediction models [9, 24, 32, 33, 45]. Formal methods and program analysis can identify some classes of configuration defects [15, 64], leading to variability-aware testing approaches (e.g., [21, 34, 35, 37, 38, 46, 55, 58, 61, 66]). The general principle is to exploit the commonalities among variants, mainly at code level. For instance, variability-aware execution [10, 37, 55] instruments an interpreter of the underlying programming language to execute the tests only once on all the variants of a configurable system. Nguyen *et al.* implemented Varex, a variability-aware PHP interpreter, to test WordPress by running code common to several variants only once [55]. Reisner *et al.* use a symbolic execution framework to evaluate how the configuration options impact the coverage of the system given a test suite [58]. Static analysis and notably type-checking has been used to analyze some properties of configurations and can scale to very large code bases such as the Linux Kernel [34, 35, 66]. Though variability-aware analysis is relevant in many engineering contexts, our interest differs and consists in concretely building a sample of (possibly distant and diverse) configurations with an unexplored approach – incremental build. Al-Hajjaji et al. [6, 7] focused on optimizing the order of a tested set of configurations, by selecting the most dissimilar configuration to the previous one to be built. However, while the authors incrementally select the configurations based on a similarity prioritization criterion, they do not perform incremental build. Sampling configurations is subject to intensive research [8, 28, 33, 62, 64]: incremental build brings new challenges *e.g.*, as empirically shown, random sampling is ineffective since the distance and differences across configurations are too important. Several empirical studies exist about the build of SPLs and configurable systems. For instance, Halin *et al.* [26] report on the endeavor to build all possible configurations of the JHipster configurable software system.

8 CONCLUSION

In this paper, we first showed that the classical way of building a large number of Linux kernel configurations does not take advantage of the incremental capabilities of Make. We then described PyroBuildS, our new approach to incrementally explore the (very large) configuration space of Linux, showing that appropriate exploration strategies trigger synergies with these caching capabilities of Make.

We evaluated the impact of these strategies on the diversity of the built configurations, showing it was possible to keep a good level of diversity as needed for proper testing while reaping the fruits of incrementality. On a total of 2520 builds, we observed a systematic gain going up to 85%, while reaching an average of 90.6% of correctness and 33% of diversity of options and 15 out of 17 subsystems that a random Linux configuration would explore in terms of activated options and subsystems.

Our PyroBuildS technique and tool thus enables building more Linux configurations for a given time (or other resource, like energy) budget. Individual contributors, testers, or janitors and continuous integration services like KernelCI and 0-day can leverage PyroBuildS to efficiently augment their configuration builds, or reduce the cost of building numerous configurations. As future work, we plan to extend PyroBuildS with Ccache to further increase the reusability of already build artefacts. We also plan to explore the benefit of having a *deny list* that would prevent some configuration options from being mutated as part of PyroBuildS.

REFERENCES

[1] [n. d.]. A fast, scalable, multi-language and extensible build system. https://bazel.build/

[2] 2021. Intel 0-day. https://01.org/lkp/documentation/0-day-test-service

[3] 2021. KernelCI. https://kernelci.org/

[4] Iago Abal, Jean Melo, Stefan Stanciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 3 (2018), 10:1–10:34.

[5] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, and Jean-Marc Jézéquel. 2019. *Learning From Thousands of Build Failures of Linux Kernel Configurations.* Technical Report. Inria ; IRISA. 1–12 pages. https://hal.inria.fr/hal-02147012

[6] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-oriented product prioritization for similarity-based product-line testing. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE).* IEEE, 34–40.

[7] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling* 18 (2019), 499–521.

[8] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling effect on performance prediction of configurable systems: A case study. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 277–288. https://doi.org/10.1145/3358960.3379137

[9] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning Software Configuration Spaces: A Systematic Literature Review. *Journal of Systems and Software* (Aug. 2021). https://doi.org/10.1016/j.jss.2021.111044

[10] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. *ACM SIGPLAN Notices* 47, 1 (jan 2012), 165.

[11] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, 356–367. https://doi.org/10.1109/MSR.2017.62

[12] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, Piscataway, NJ, USA, 356–367.

[13] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. 2017. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering* 22, 6 (2017), 3117–3148. https://doi.org/10.1007/s10664-017-9510-8

[14] Qi Cao, Ruiyin Wen, and Shane McIntosh. 2017. Forecasting the duration of incremental build jobs. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 524–528. https://doi.org/10.1109/ICSME.2017.34

[15] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (aug 2013), 1069–1089.

[16] Máté Cserép and Anett Fekete. 2020. Integration of Incremental Build Systems Into Software Comprehension Tools.. In *ICAI*. 85–93. http://ceur-ws.org/Vol-2650/paper10.pdf

[17] Jack Edge. 2020. The costs of continuous integration. https://lwn.net/Articles/813767/

[18] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. *ACM Sigplan Notices* 50, 10 (2015), 89–106.

[19] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265. https://doi.org/10.1002/spe.4380090402 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380090402

[20] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265.

[21] Stefan Fischer, Rudolf Ramler, Claus Klammer, and Rick Rabiser. 2021. Testing of Highly Configurable Cyber-Physical Systems – A Multiple Case Study. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems* (Krems, Austria) *(VaMoS'21)*. Association for Computing Machinery, New York, NY, USA, Article 19, 10 pages. https://doi.org/10.1145/3442391.3442411

[22] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice* (Virtual Event, Spain) *(ICSE-SEIP '21)*. IEEE Press, 91–100. https://doi.org/10.1109/ICSE-SEIP52600.2021.00018

[23] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. 2020. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* (2020), 1–1.

[24] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[25] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2018. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* 24, 2 (jul 2018), 674–717. https://doi.org/10.1007/s10664-018-9635-4

[26] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empir. Softw. Eng.* 24, 2 (2019), 674–717. https://doi.org/10.1007/s10664-018-9635-4

[27] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. 2015. Incremental computation with names. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct 2015).

[28] Ruben Heradio, David Fernandez-Amoros, José A Galindo, David Benavides, and Don Batory. 2022. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering* 27, 2 (2022), 44.

[29] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[30] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 197–207.

[31] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 426–437.

[32] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: an exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 497–508.

[33] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 71–82.

[34] Christian Kastner and Sven Apel. 2008. Type-checking software product lines-a formal approach. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - ASE '08*. IEEE, 258–267.

[35] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development* (Eindhoven, The Netherlands) *(FOSD '10)*. ACM, New York, NY, USA, 25–32.

[36] Michael Kerrisk. 2012. KS2012: Kernel build/boot testing. https://lwn.net/Articles/514278/.

[37] Chang Hwan Peter Kim, Don S Batory, and Sarfraz Khurshid. 2011. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development (AOSD '11)*. ACM, 57–68.

[38] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems - ESEC/FSE '13. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 257–267.

[39] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 76–86. https://doi.org/10.1145/3238147.3238196

[40] Gabriël Konat, Roelof Sol, Sebastian Erdweg, and Eelco Visser. [n. d.]. Precise, Efficient, and Expressive Incremental Build Scripts with PIE. ([n. d.]).

[41] Julia Lawall and Gilles Muller. 2017. JMake: Dependable Compilation for Kernel Janitors. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 357–366.

[42] Julia Lawall and Gilles Muller. 2017. JMake: Dependable Compilation for Kernel Janitors. In *The 47th IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE/IFIP, Denver,Colorado, United States. https://doi.org/10.1109/DSN.2017.62

[43] Carlene Lebeuf, Elena Voyloshnikova, Kim Herzig, and Margaret-Anne Storey. 2018. Understanding, Debugging, and Optimizing Distributed Software Builds: A Design Study. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 496–507. https://doi.org/10.1109/ICSME.2018.00060

[44] Carlene Lebeuf, Elena Voyloshnikova, Kim Herzig, and Margaret-Anne Storey. 2018. Understanding, debugging, and optimizing distributed software builds: A design study. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 496–507.

[45] Luc Lesoil, Mathieu Acher, Xhevahire Tërnava, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. The interplay of compile-time and run-time options for performance prediction. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, Mohammad Mousavi and Pierre-Yves Schobbens (Eds.). ACM, 100–111.

[46] Jackson A. Prado Lima, Willian Douglas Ferrari Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 31:1–31:11.

[47] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2022. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Trans. Software Eng.* 48, 11 (2022), 4274–4290. https://doi.org/10.1109/TSE.2021.3116768

[48] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2022. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4274–4290. https://doi.org/10.1109/TSE.2021.3116768

[49] Guillaume Maudoux and Kim Mens. 2017. Bringing incremental builds to continuous integration. In *Proc. 10th Seminar Series Advanced Techniques & Tools for Software Evolution*. 1–6.

[50] Guillaume Maudoux and Kim Mens. 2018. Correct, efficient, and tailored: The future of build systems. *IEEE Software* 35, 2 (2018), 32–37.

[51] Guillaume Maudoux and Kim Mens. 2019. Lessons and Pitfalls in Building Firefox with Tup.. In *SATToSE*.

[52] Shane Mcintosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. 2015. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Softw. Engg.* 20, 6 (dec 2015), 1587–1633. https://doi.org/10.1007/s10664-014-9324-x

[53] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wąsowski. 2016. A Quantitative Analysis of Variability Warnings in Linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems* (Salvador, Brazil) *(VaMoS '16)*. ACM, 3–8.

[54] Neil Mitchell. 2012. Shake before building. *ACM SIGPLAN Notices* 47 (10 2012), 55. https://doi.org/10.1145/2398856.2364538

[55] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering - ICSE '14*. ACM, 907–918.

[56] Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. 2022. Towards incremental build of software configurations. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 101–105.

[57] Georges Aaron Randrianaina, Xhevahire Tërnava, Djamel Eddine Khelladi, and Mathieu Acher. 2022. On the benefits and limits of incremental build of software configurations: an exploratory study. In *Proceedings of the 44th International Conference on Software Engineering*. 1584–1596.

[58] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. 2010. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10, Vol. 1)*. ACM Press, 445.

[59] Dan Rue. 2021. Portable and reproducible kernel builds with TuxMake. https://lwn.net/Articles/841624/.

[60] Robert W. Schwanke and Gail E. Kaiser. 1988. Smarter Recompilation. *ACM Trans. Program. Lang. Syst.* 10, 4 (Oct. 1988), 627–632.

[61] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (LNCS, Vol. 7212)*. Springer, 270–284.

[62] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 632–642.

[63] Tuxmake Team. Online; accessed 2023. TuxMake. https://tuxmake.org/.

[64] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 6:1–6:45.

[65] Walter F. Tichy. 1986. Smart Recompilation. *ACM Trans. Program. Lang. Syst.* 8, 3 (June 1986), 273–291. https://doi.org/10.1145/5956.5959

[66] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33.

[67] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[68] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 60–71.