Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times

NECIP FAZIL YILDIRAN, University of Central Florida, USA
JEHO OH, University of Texas, USA
JULIA LAWALL, Inria, France
PAUL GAZZILLO, University of Central Florida, USA

The Linux kernel is highly-configurable, with a build system that takes a configuration file as input and automatically tailors the source code accordingly. Configurability, however, complicates testing, because different configuration options lead to the inclusion of different code fragments. With thousands of patches received per month, Linux kernel maintainers employ extensive automated continuous integration testing. To attempt patch coverage, i.e., taking all changed lines into account, current approaches either use configuration files that maximize total statement coverage or use multiple randomly-generated configuration files, both of which incur high build times without guaranteeing patch coverage. To achieve patch coverage without exploding build times, we propose krepair, which automatically repairs configuration files that are fast-building but have poor patch coverage to achieve high patch coverage with little effect on build times. krepair works by discovering a small set of changes to a configuration file that will ensure patch coverage, preserving most of the original configuration file's settings. Our evaluation shows that, when applied to configuration files with poor patch coverage on a statistically-significant sample of recent Linux kernel patches, krepair achieves nearly complete patch coverage, 98.5% on average, while changing less than 1.53% of the original default configuration file in 99% of patches, which keeps build times 10.5x faster than maximal configuration files.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Software maintenance tools**; *Software testing and debugging*.

Additional Key Words and Phrases: software configuration, build systems, static analysis

ACM Reference Format:
Necip Fazıl Yıldıran, Jeho Oh, Julia Lawall, and Paul Gazzillo. 2024. Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times. *Proc. ACM Softw. Eng.* 1, FSE, Article 20 (July 2024), 23 pages. https://doi.org/10.1145/3643746

1 INTRODUCTION

The Linux kernel is a prototypical example of a highly-configurable system. Users can adapt the Linux kernel to virtually endless combinations of hardware and software requirements by simply selecting configuration options, with no additional programming [11, 32, 39]. This high degree of configurability allows the Linux kernel to be used in very diverse environments, including all of the top 500 supercomputers [7], 40% of servers [62], and the majority of Internet-of-Things devices [27]. Nevertheless, this degree of configurability complicates testing, because different configuration

Authors' addresses: Necip Fazıl Yıldıran, University of Central Florida, Orlando, USA, yildiran@knights.ucf.edu; Jeho Oh, University of Texas, Austin, USA, jeho.oh@utexas.edu; Julia Lawall, Inria, Paris, France, julia.lawall@inria.fr; Paul Gazzillo, University of Central Florida, Orlando, USA, paul.gazzillo@ucf.edu.

ACM 2994-970X/2024/7-ART20
https://doi.org/10.1145/3643746

options lead to the inclusion of different code fragments and thus different runtime behaviors. Configurability is especially challenging when the software is rapidly changing, as changes must be validated with respect to software configurations that actually do include the changed code. The Linux kernel receives thousands of patches per month, and automated continuous integration testing is extensively used to cope with this rate of change. To try to achieve *patch coverage*, i.e., that all changed lines are taken into account, current continuous integration testing approaches either use configuration files that select as many configuration options as possible (for the Linux kernel, `make allyesconfig`) or use multiple randomly generated configuration files (`make randconfig`), both of which lead to high build times without guaranteeing success.

State-of-the-art approaches to generating configuration files target increasing feature-interaction coverage or statement coverage, but are not designed for patch coverage. Approaches targeting feature-interaction coverage systematically test many combinations of features [17, 64], e.g., all pairs of features or all triples of features. But such approaches do not scale to testing software with large numbers of configuration options [50], and even for smaller systems they generate thousands of configuration files, requiring enormous resources to build and test continuously. And feature-interaction coverage does not guarantee patch coverage [50].

Statement-coverage approaches, in contrast, seek to cover the most code with the fewest configuration files [50], which results in high build times, and can still fail to cover patches. Indeed, the configuration file obtained using `allyesconfig` takes more than ten times longer to build than the Linux kernel's default configuration (obtained using `make defconfig`), and the very large size of `allyesconfig` means that it is not suitable for booting on some machines [48]. Statement-coverage approaches are thus highly resource intensive for continuous integration testing, which needs to test hundreds of patches several times a day. Moreover, when examining build-test reports from the Linux kernel 0-day build testing service [22], the large majority (63%) are randomly-chosen configurations (`randconfig`) compared to many fewer reports of `allyesconfig` (15%) (Section 2).

To achieve patch coverage while preserving the original configuration file and its build times, we propose to construct configurations that are targeted to the specific changes found in a given patch. We introduce a new algorithm, called krepair, to solve the problem of generating configuration files for efficient continuous integrating testing of highly-configurable software. It works by *repairing* a user-provided configuration file to ensure patch coverage without resorting to maximal configurations, and preserves most of the original configuration file's settings. For instance, Linux's small default configuration (`make defconfig`) rarely covers the code in patches but builds relatively quickly. After repairing by krepair for a given patch, the repaired default configuration almost always covers the patch with only marginal additional build time, while in most cases, krepair only takes a few minutes to find a covering configuration file. This approach thus retrofits existing continuous integration testing for highly-configurable software to provide high patch coverage with little additional cost, since it repairs any existing configuration files already used or generated by testers.

krepair works by discovering, using automated reasoning, a small set of changes to a configuration file that will ensure patch coverage. It first collects a set of patch coverage constraints for all changed lines of code. This step draws on statement-coverage approaches [66], using existing line coverage constraint extractors [31, 33, 53] and building on the VAMPYR algorithm [66] to find a set of patch-covering constraints for a given patch. The challenge for krepair is to combine these constraints with a test platform's existing configuration file, which often has low patch coverage but fast build times, without introducing contradictions; such contradictions indeed often arise because of the many complex dependencies among the Linux kernel configuration options. To overcome this challenge, krepair uses an automated theorem prover to detect which configuration file settings cause contradictions with the patch coverage constraints and removes these settings, little by

little, until the resulting configuration file satisfies the constraints. Then, it repopulates missing configuration settings by querying the prover for a solution that preserves the patch constraints. Since automated theorem proving is expensive, krepair employs several optimizations to reduce the number of calls to the prover. When there are mutually-exclusive changes in a patch, i.e., no one configuration file can cover the patch, krepair detects this and generates a small set of repaired configuration files that collectively cover the whole patch. In practice, 97% of patches we have tested produced just one configuration file. We have implemented krepair in Python as a command-line tool that works on the Linux build system, a build system that is also used by other low-level systems software, such as BusyBox [1] and coreboot [21].

We evaluate krepair by measuring how well it ensures that configuration files cover patches while keeping the build times fast enough for continuous integration testing. We use a statistically significant sample of patches from one full of year of about 71,000 patches resulting in a sample of 507 patches. We quantify patch coverage as the number of removed or added lines included by the build configuration divided by the total number of removed or added lines in the patchfile. To measure patch coverage, we intercept the build system to check whether the files and lines of the patch have been included in the build, then we compare the coverage of each patch before and after repair. To measure build time, we build the entire kernel from scratch on an AMD EPYC compute server using the configuration file and record the wall clock time. We compare the patch coverage and build time against the Linux default configuration and its statement maximizing configuration file `allyesconfig`. The set of configuration files generated by krepair achieves 98.5% patch coverage on average, compared to 21.7% for the default configuration file, `defconfig`. krepair's configuration files even have higher coverage than `allyesconfig` on average, which covers 88.5%. But krepair's set of `defconfig`-based configuration files are 10.5x faster to build than `allyesconfig` and comparable in build time to `defconfig`. In short, krepair achieves the patch coverage of statement-covering approaches without the cost in resources, taking only a small fraction of the build time, while krepair itself finds a patch-covering configuration file in a few minutes in most cases.

krepair achieves fast build times, because it preserves most settings from its input configuration file while still covering patches. We show that in 99% of patches, it only changes 1.5% or fewer configuration options to achieve patch coverage. Additionally, since random configuration testing is used by some of the largest industrial continuous testing infrastructures [20, 22], we also measure how much patch coverage such testing can achieve. We show that a single random configuration file only achieves 29.2% patch coverage on average, while adding more random configuration files has diminishing coverage returns, plateauing at around 75% with 10 random configuration files. Moreover, using multiple random configuration files to achieve patch coverage increases build time, since each random configuration file needs to be built individually for testing. We even find some build errors that were overlooked when the patches were integrated into the Linux kernel, showing that krepair complements existing testing approaches. We describe the 18 build errors found by repaired configuration files, including 2 errors that had not yet been fixed, one of which has already had our patch accepted by the Linux developers.

This paper makes the following contributions:

- An algorithm that automatically repairs existing configuration files to cover patches with little effect on build times (Section 3).
- The implementation of krepair, with caching to improve performance (Section 4).
- An evaluation of krepair for patch coverage, build times, and configuration preservation, with a comparison to statement-coverage maximizing and random configuration-file generation approaches (Section 5).

2 BACKGROUND

When testing tools do not ensure patch coverage, they cannot exhaustively test changes to the code. For instance, `syzbot` performs continuous testing of the Linux kernel using the `syzkaller` [35] fuzz-tester and was responsible for the majority of credited reports to the release v4.9 [19]. But it relies on a small, fixed set of configuration files with the configuration options necessary to run syzbot [35]. These configurations provide no assurance that code in new patches gets compiled before testing. A memory leak in the Linux kernel [63] that `syzkaller` can detect [2] remained in the kernel for months, because the configuration option controlling inclusion of the buggy code was not enabled. `syzkaller` only found the bug months later, after the configuration option happened to be included by the default configuration in a later version of Linux [3].

To understand the configurations that the Linux kernel developer community considers to be useful to test, we study the e-mail history[1] of the Linux kernel 0-day build testing service [22], a continuous integration service developed by Intel. This service performs both performance tests and build tests (including running various static analysis tools), and mails reports to patch developers on any detected regressions. Accordingly, we only have access to information about the configurations in which regressions were detected, but these are also the configurations that have been the most useful. We downloaded all of the available build-test messages from October 1, 2019 through August 27, 2022, resulting in 36,115 reports from the 0-day service containing configuration files. Of these, the largest proportion are created using `make randconfig`, amounting to 22,702 configurations, or 63% of the total. This is followed by `make allyesconfig` at 5,551 (15%), `make defconfig` at 2,708 (7%), and `make allmodconfig` (analogous to `allyesconfig`, but trying to select as many modules as possible) at 2,446 (7%). The remaining 8% were miscellaneous configuration files. These results indicate that while the 0-day service does find the statement-coverage targeting configuration `allyesconfig` to be useful, most of its results are derived from `make randconfig`, which typically results in much smaller configurations. But, as we show (Section 5), `randconfig` provides little guarantee of patch coverage, even when run many times.

To help understand the challenges of performing continuous testing of highly-configurable software, we first overview the space of approaches to generating configuration files for testing such software. Then we describe the Linux kernel build system, particularly focusing on the Kconfig language, and present a patch that raises configuration challenges. We finally consider how the Linux kernel build-system design impacts the problem of achieving build coverage of the lines affected by a given patch.

2.1 Configuration Testing Approaches

Fundamentally, the first challenge of testing of changes to highly-configurable software is to ensure that changes are not excluded from the tested binary, i.e., that the patched lines of code are compiled by the build system into the binary. Performing continuous testing requires finding configurations that cover patches fast enough so that the testing infrastructure can keep up with the rapid pace of changes, which in Linux kernel development means hundreds of patches a day. There is a trade-off in build time, which can require hours of processor time, and patch coverage. Table 1 compares state-of-the-art configuration file generation techniques along these two axes.

Configuration file generation approaches that cover many feature interactions require generating many configuration files. For instance, t-wise sampling ensures that each combination of t configuration options is covered by some configuration file. For 2-wise sampling, each of the four possible combinations of two options set to on or off needs to be covered by some configuration file for all pairs of configuration options. As the Linux kernel has over 15,000 configuration options,

[1] https://web.archive.org/web/20221023104058/https://lists.01.org/hyperkitty/

Table 1. Comparing configuration testing approaches for use in continuous testing.

		Patch Coverage	
		Lower	Higher
Build Time	Slower	t-wise [50, 64] Combinatorial testing [17]	`allyesconfig` [4] VAMPYR [66]
	Faster	`randconfig` [4] `defconfig` [4]	**krepair**

2-wise coverage would require considering a very large number of pairs, implying that even the most efficient algorithms cannot cover all interactions for the Linux kernel [50]. Such approaches are not designed for the problem of efficient patch coverage, but rather to test feature interactions, so repurposing them for patch coverage means very resource-intensive build times, due to the many configuration files needed.

Random configuration-file generation for testing is popular in industrial testing tools [20, 22], because each configuration file is much faster than using a statement-covering configuration file. But our evaluation shows that individual randomly-generated configuration files have a low chance of covering patches. Industrial tools compensate for a lack of coverage by generating multiple random configuration files, dozens in some cases [15, 22]. But as our evaluation (Section 5) also shows, adding more random configuration files has diminishing returns for patch coverage. The tenth random configuration file adds less than a percent of additional coverage, and ten configuration file collectively only achieve 74.6% patch coverage on average. Adding more configuration files also inflates build time, because the total build time is proportional to the number of configuration files used.

Statement-maximizing approaches, such as `allyesconfig`, do have high patch coverage, since they attempt to cover as much of the source code as possible in one or a few configuration files. But this good patch coverage, 88.5% on average, comes at the cost of much slower build times, around four hours on average on a typical development workstation, compared to the default configuration or to krepair's configuration files, which take only around 20 minutes on average. VAMPYR is a state-of-the-art statement coverage approach that improves on `allyesconfig` [66]. Based on presence conditions for all of the statements in a targeted code base, it employs a SAT solver to find a set of configuration settings to cover the lines not covered by `allyesconfig`, and then exploits the Linux kernel's `make olddefconfig` to extend each resulting set of configuration settings with default values to form a complete configuration. VAMPYR thus produces a set of configuration files covering more than `allyesconfig`, but at the cost of even slower build times, since it requires building at least for `allyesconfig` as well as for its generated configuration files, and the set of generated configuration files is not limited to what is needed for a specific patch. While evaluated for maximal statement coverage, VAMPYR's constraint covering approach can also be applied to only cover a patch's line constraints [67]. krepair builds on this constraint coverage approach by adding configuration repair, which automatically reconciles any existing configuration file, even very small ones, with patch coverage constraints, resolving the trade-off in build time and patch coverage by simultaneously enforcing patch coverage constraints and preserving most of the original configuration file's settings.

Fig. 1. Build system components handling configuration.

2.2 Linux Kernel Configuration

To help explain why the problem of finding an efficient, patch-covering configuration file is so difficult, let us first look at how the build system defines and uses configuration options. The Linux kernel build system takes a configuration file as input and determines what files and lines of source code to compile into the kernel binary. Figure 1 shows the relevant components of the build system.

The first component is a collection of Kconfig files spread across the Linux kernel code base that formally describe the constraints on the configuration options relevant to each subsystem. Kconfig is used to validate the input configuration file. Figure 2 shows a patch[2] (Figure 2c) and Kconfig specifications (Figure 2a) for the options controlling the patched code. The option with the simplest constraints is `PM` (Figure 2a, lines 11-12), which implicitly determines the value of the configuration variable `CONFIG_PM`. `PM` is declared as a Boolean (yes or no). The associated prompt "Device power ..." indicates that the user will be asked with this prompt for the desired value. `ARCH_EXYNOS4` (line 16) is declared similarly, but it has a default value of `y` (yes, line 18). In contrast, the constraints on `ARM_GIC`, `ARM_GIC_PM`, and `GIC_NON_BANKED`, indicate that, while these options are also Booleans, they cannot be specified directly by the user, as no prompt is provided. Such configuration options may be declared to *depend on* the selection of another configuration option or can be *selected* by some other option. `ARM_GIC_PM` depends on `PM` (line 5), and if it is selected, then it also selects `ARM_GIC` (line 6). Likewise, selecting `ARCH_EXYNOS4` selects `GIC_NON_BANKED`. Finally, further constraints can be expressed using conditionals (e.g., `if ... endif`), as illustrated on lines 15-20. A provided configuration file is checked to respect the various constraints specified by the Kconfig files, and is enhanced with any selected or dependent configuration options based on the options selected in the configuration file. The result is a configuration that controls the rest of the build process.

The second component is the collection of Kbuild Makefiles spread across the Linux kernel code base that describe how to build and link the various subsystems. As illustrated in Figure 2b, these Makefiles use the configuration variables to determine what files to include in the generated kernel. For example, `irq-gic.c` is only compiled and included if `CONFIG_ARM_GIC` is set.

Finally, the third component is the source code itself. Illustrated by line 4 of the patch (Figure 2c), source code may refer to configuration variables directly via `#ifdefs`. These `#ifdefs` select the specific lines of code that will be included in the compiled kernel. Changing the configuration file changes requires rebuilding the whole kernel, i.e., `make clean`, because make has no visibility over the `#ifdefs` used within C files.

2.3 Motivating Example

We next look at the same configuration constraints from the point of view of covering the changed lines of a patch. The patch in Figure 2c affects the file `drivers/irqchip/irq-gic.c`. It is formatted in the standard unified diff format [49], in which the `-` prefix (lines 5, 8, 12, and 16) indicates a line to remove and the `+` prefix (lines 6, 9, 13, and 17) indicates a line to add. To cover the patch, a configuration must cause the modified file to be included in the build and ensure that all the changed lines are included in the build.

[2] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8594c3b85171b6f68e34e07b533ec2f1bf7fb065

```
config ARM_GIC
     bool
config ARM_GIC_PM
     bool
     depends on PM
     select ARM_GIC
config GIC_NON_BANKED
     bool

// from kernel/power/Kconfig
config PM
     bool "Device power management core functionality"

// from arch/arm/mach-exynos/Kconfig
if ARCH_EXYNOS
config ARCH_EXYNOS4
     bool "Samsung Exynos4"
     default y
     select GIC_NON_BANKED
endif
```

(a) Kconfig specifications for options controlling the patched code, showing a few of the many dependencies.

```
// from drivers/irqchip/Makefile
obj-$(CONFIG_ARM_GIC)   += irq-gic.o
```

(b) Relevant build specifications for the patched file.

```
--- a/drivers/irqchip/irq-gic.c
+++ b/drivers/irqchip/irq-gic.c
@@ -127,35 +124,27
 #ifdef CONFIG_GIC_NON_BANKED
-static void *gic_get_common_base(union gic_base *base)
+static void enable_frankengic(void)
 {
- return base->common_base;
+ static_branch_enable(&frankengic_key);
 }
 #else
-#define gic_set_base_accessor(d, f)
+#define enable_frankengic() do   while(0)
 #endif
@@ -1165,7 +1149,7
- gic_set_base_accessor(gic, gic_get_percpu_base);
+ enable_frankengic();
```

(c) Hunks from the patch to Linux source, edited for brevity.

Fig. 2. An example patch to the Linux source and the configuration specifications controlling its buildability.

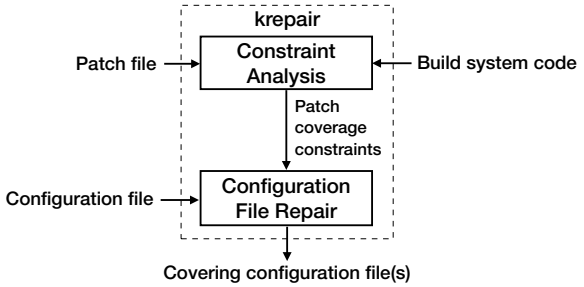

Fig. 3. Workflow of krepair.

We start with the file. Checking the Makefile in the same directory (Figure 2b) shows that building the file requires selecting the `ARM_GIC` configuration option. As previously noted, the user cannot select this option directly; instead, it is necessary to trace across multiple Kconfig files to discover that this option can be selected by the option `ARM_GIC_PM`. The latter option also cannot be selected directly but depends on `PM`.

We next turn to the lines changed within the file. Indeed, simply ensuring that the build includes the file does not ensure that the build includes the changed lines, because some of these lines are under an `#ifdef`. The `#ifdef` involves the configuration option `GIC_NON_BANKED`; another search is needed to identify a selectable option that will cause this option to be selected. But for this patch it is not sufficient to select `GIC_NON_BANKED`, because the patch modifies code under the `#else` as well. The changes are thus mutually-exclusive and therefore covering all the changed lines requires at least two configuration files, one that selects `GIC_NON_BANKED`, and another that does not.

Still, even with all of the above collected information, the task of creating usable covering configuration files is not complete. There are thousands of other options that need to be assigned, some of which may even influence whether the two identified options themselves are selectable. Test cases that involve specific kernel features may introduce more configuration conflicts.

Assessment. The challenges in finding one or more configurations that cover a patch, as illustrated by the motivating example, come from the design of the build system. Indeed, the build system is designed to take a configuration file and determine what lines of code to build, but not the other way around. We can think of the build system as defining logical constraints on each line of source code [53], and a configuration file as one solution to those constraints. Determining the inverse, i.e., what configuration files build a certain line of code, is equivalent to a satisfiability problem, which is computationally expensive in the general case. Finding what repairs to make to an existing configuration file requires determining what options directly control the patched lines, then reconciling those options with their dependencies and the settings in the existing file as much as possible, favoring patch coverage when settings in the existing file contradict the patch coverage constraints.

3 THE KREPAIR ALGORITHM

krepair automatically repairs an existing configuration file to ensure complete coverage of buildable code in a given patch. It works in two steps: (1) discover a covering set of constraints for the lines changed (removed or added) by a patch and (2) find a set of changes to the existing configuration file that will satisfy these constraints.

Figure 3 shows the high-level krepair workflow. Constraint Analysis takes as input the patch itself and the source code of the build system. The output of the Constraint Analysis is a set of patch coverage constraints found by statically analyzing the build system code. Configuration-File Repair

Algorithm 1 KREPAIR(patchlines, configuration, commitid) - Repair an existing configuration file to cover the patch.

Input: A list of (file, line) pairs in patchlines from the patch and an existing configuration file configuration.
Output: A set of repaired configuration files that cover that patch.

```
1:  function KREPAIR(patchlines, configuration, commitid)
2:    allrepaired ← ∅
3:    do
4:      current ← true
5:      covered ← ∅
6:      for file,line ∈ patchlines do
7:        constraints ← GETCONSTRAINT(file, line, commitid)
8:        if isSAT(current ∧ constraints) then
9:          current ← current ∧ constraints
10:         covered ← covered ∪ {(file,line)}
11:       end if
12:     end for
13:     if covered ≠ ∅ then
14:       repaired ← REPAIR(configuration, current)
15:       allrepaired ← allrepaired ∪ {repaired}
16:       patchlines ← patchlines − covered
17:     end if
18:   while patchlines ≠ ∅ ∧ covered ≠ ∅
19:   return allrepaired
20: end function
21: function REPAIR(configuration, constraint)
22:   repair ← configuration
23:   repeat
24:     unsat ← UNSATCORE(repair ∧ constraint)
25:     repair ← repair − unsat
26:   until unsat = ∅
27:   return SATSOLVE(repair ∧ constraint)
28: end function
```

then takes as input the existing configuration file to repair and produces one or more configuration files that are close to the input file but modified to cover the patch.

Algorithm 1 describes krepair in pseudo-code. The algorithm takes as input a list of pairs of the file name and line number of those lines that are changed (added or removed) by a patch file. The line number of an added line reflects its position after applying the patch. The line number of a sequence of consecutive removed lines is the number of the line just preceding the removal after applying the patch. The second input is the configuration file that needs repair. The third input is the version of the code, as a commitid, that has had the patch applied to it. The output is a *set* of configuration files, since some patches may touch lines depending on mutually-exclusive configuration values. For example, the patch in Figure 2c changes both arms of an `#ifdef`. Therefore, our algorithm cannot just conjoin all patch line constraints. Instead, it tries to find a small set of satisfiable configurations that, together, cover the entire patch. In practice, we find a single configuration for 97% of patches, five or fewer for more than 99% of patches, and 23 in the worst case.

3.1 Constraint Analysis

The algorithm first performs constraint analysis to partition the set of constraints controlling each patched line into subsets of constraints that do not contradict each other. For this, it iterates repeatedly over the set of patched lines (lines 3–18). Each iteration starts with an empty constraint (line 4). krepair then iterates over each (file, line) pair (lines 6–12), and greedily tries to cover as many pairs as possible within a single constraint (`current`). This part of the algorithm draws from VAMPYR [66], which achieves statement coverage by finding covering constraints. Each (file, line) pair's constraint is provided by third-party tools that analyze the build system (GetConstraint on line 7).

If the (file, line) pair's constraint does not contradict the constraint accumulated so far (line 8), the algorithm updates the current constraint with that of the (file, line) pair (line 9). It keeps track of which (file, line) pairs have already been accumulated (line 10), so that the algorithm will remove them from the set of candidates (line 16). Once as many (file, line) pairs as possible have been accumulated, the algorithm repairs the configuration file according to the accumulated current constraint (line 14) and adds the result to a collection of repaired configurations (line 15). krepair stops trying to cover patch lines when either there are no more patch lines left to cover, or when it finds that no other patch lines can be covered by any configuration (line 18). The latter happens when (file, line) is unconfigurable, e.g., if it is configurable in another architecture or dependent on dead configuration options.

Optimizations. This algorithm relies on third-party constraint collection tools (line 7) and satisfiability solving to partition the set of patch line constraints (line 8), both of which are computationally expensive. We make three optimizations. The first optimization checks whether the patched line is inside any #ifdef block. If not, then there is no need to collect constraints from the source file; the line is always included if the file is included. The second optimization checks whether a changed line is within the same set of #ifdef blocks as an already-seen changed line. In this case, there is no need to collect constraints for the current changed line. These optimizations reduce the number of calls to the constraint-finding tool. The third optimization targets nested #ifdef blocks. In this case, if the constraints for the inner block are satisfiable, then the constraints for the outer block must also be satisfiable. Conversely, if the constraints for the outer block are not satisfiable, then the constraints for the inner block are also not satisfiable. This optimization reduces the number of calls to the satisfiability checker.

3.2 Configuration-File Repair

The repair part distinguishes krepair from previous coverage approaches by automatically tailoring an existing configuration file so that it is patch covering without much change to the configuration. The Repair function in Algorithm 1 repairs the configuration file according to the given patch coverage constraints (lines 21–28). It takes a configuration file and a constraint from krepair's constraint analysis and returns a configuration file close to the input file, but modified to satisfy the patch coverage constraints. The repair algorithm repeatedly checks the configuration file against the patch coverage constraints and gradually removes configuration option settings until the configuration file satisfies the constraints. Then, it repopulates any removed configuration option settings by taking a satisfying solution to the constraints.

The key trade-off in the repair algorithm is the computational complexity of finding the right settings to remove to satisfy the constraints of the patch while limiting the number of removals to keep the configuration file similar to the original. A naive optimal algorithm for finding the minimal number of removals would be to check all combinations of setting removals against the constraints. But this is prohibitively expensive, having an exponential number of satisfiability

checks, i.e., the power set of thousands of configuration options. A faster approach would be to remove some arbitrary number of options, check satisfiability after removing them, and repeat as needed. But this approach might unnecessarily remove options that do not conflict with the patch coverage constraints.

Our algorithm has the better performance of the faster approach, while homing in on options that are preventing satisfiability more quickly. For this, it repurposes feedback from the automated theorem prover, called an *unsatisfiable core*, to guide what settings to remove. The unsatisfiable core is a (not necessarily minimal) subset of the original clauses that is still unsatisfiable [47]. By only removing settings in the unsatisfiable core, Repair gradually finds a subset of the configuration file options preventing satisfiability (lines 23-26). Each new satisfiability check produces a new unsatisfiable core (line 24), which provides new removal candidates (line 25). Since Krepair only passes satisfiable patch coverage constraints to Repair, the unsatisfiable core always contains at least one configuration option as long as `configuration` $\wedge$ `constraint` is unsatisfiable, guaranteeing termination. Finally, the missing constraints are repopulated by finding a satisfying solution to the reduced configuration file under the patch coverage constraints (line 27).

4 IMPLEMENTATION

krepair is implemented as a command-line utility in ~3000 lines of python code. It relies on third-party constraint-finding tools [31, 33, 53] for GetConstraint. krepair runs from the root of a Linux kernel source tree, so it can identify the build system source files from its working directory. It takes a patchfile and an existing configuration file on the command-line and produces output configuration files in the format expected by the build system.

4.1 Processing Patch Files

Linux kernel patches are represented in the unified diff format [49]. krepair parses a patch using `whatthepatch` [5] and converts the patch into a set of after-patch (file, line) pairs. krepair is line-oriented, so a patch that adds a new file requires checking coverage of all lines in the file. krepair does nothing when a file is simply renamed, as no lines are changed. It also does nothing for removed files, as they are no longer buildable after the patch and therefore have no build constraints. Removed lines, however, are considered changes just like added lines, since we can identify the configuration constraints affected by both by gathering the constraints for the file and any `#ifdef` that contain them.

The build system only explicitly defines constraints for compilation units. Therefore, krepair provides limited support for patches to C header files, since headers are only included indirectly by other source files. We use a simple heuristic to find covering constraints for lines in header files: krepair assumes the header file has the same build constraints as the compilation units modified by the patch. While this heuristic has some success in our evaluation, it means that patches that only modify header files are not supported, which we only encountered in 2% of patches in our evaluation.

4.2 Collecting Build Constraints

krepair uses third-party static analysis tools to collect build system constraints from each of the three build components: `kclause` [53] for Kconfig configuration specification constraints, `kmax` [31] for Kbuild Makefile constraints, and `SuperC` [33] for preprocessor-level constraints in C source code. Both `kclause` and `kmax` represent constraints in the SMT-LIBv2 format [12], a standard representation of logical formulas for automated theorem provers. `SuperC`, however, was not originally designed for reporting C preprocessor constraints, although its preprocessor collects them internally. We forked the `SuperC` source code and added support for exporting the

configuration constraints of all `#ifdef` ranges and their constraints from a given source file in the SMT-LIBv2 format.

krepair's constraint collection module interfaces with all three tools, providing python wrappers around each to implement the GetConstraints function from Algorithm 1. For a given (file, line) pair, GetConstraint collects the results from each of the three tools, and then conjoins them into a single constraint for the line.

4.3 Improving Performance

Our algorithm only needs access to per-line build constraints for a given patch. But the tools we use to collect constraints were designed to run on the entire build system source files. For instance, `kclause` takes as input the entire 140,000+ lines of Kconfig specification and converts it to about 60,000 logical clauses all at once. The tools are thus time-consuming to run, with `kclause` typically taking 2-3 minutes, `kmax` 10-15 minutes, and `SuperC` about a minute or less on commodity hardware, all to get a single line's constraints, depending on the Linux kernel version and the target architecture. To reduce the cost of constraint collection for a single patch, we modified the `kmax` interface to support collecting per-file constraints on-demand. We also modified `SuperC` to emit per-line constraints for the entire source file.

The Kconfig configuration specification is a single large constraint for each of the supported architectures. Since the Kconfig specifications do not change with every patch, krepair manages an on-disk cache of Kconfig constraints indexed by a unique identifier of the Kconfig version that will be reused as long as the Kconfig specification has not changed. Similarly, Kbuild Makefiles, which define constraints on source files, only need to be collected once for a given file until the Makefile source code changes.

4.4 Implementing Repair

As with build constraints, the configuration file is represented as a set of SMT-LIBv2 constraints. krepair has functions to parse configuration files into constraints and to deparse satisfying solutions to constraints back into the configuration file format. The implementation of the repair algorithm (Algorithm 1) uses the z3 [23] automated theorem prover to check satisfiability (isSAT), find an unsatisfiable core (UnSATCore), and get a satisfying solution (SATSolver). z3 is not guaranteed to provide a minimal unsatisfiable core, but we find that the resulting cores are small enough that our repairs incur little change, less than 2.23% change for 99% of repairs.

5 EMPIRICAL EVALUATION

We evaluate krepair on a representative sample of Linux kernel patches, measuring how well it ensures that configuration files cover patches while keeping the build times fast enough for continuous testing. We study patches from the Linux kernel, because it is large, highly-configurable, very actively developed, and used in critical computing infrastructure.

5.1 Experimental Setup

Sampling patches. We have taken a random sample of 507 patches out of the approximately 71,000 patches from one recent whole year (2021/09/19–2022/09/18) of Linux kernel development, which provides a 5% margin of error with a 98% confidence level. We performed sampling by cloning the mainline Linux kernel repository [4] and using `git log` on the above date range. We exclude merge commits, which typically do not change code, and include only patches to buildable kernel source files, which excludes documentation text files, example programs, build tools, and header files. Such files are not covered by any configuration file, since they do not get compiled and linked

into the kernel binary, although header files may be indirectly covered when they are included in other kernel source files.

Configuration file collection. Our baseline for a fast-building configuration file is the *default configuration file* distributed with the kernel source for the x86 platform. This configuration file, created with `make defconfig`, is a small, quick-to-build kernel configuration frequently recommended as a starting point for building the kernel [35, 74] and frequently used in testing [22, 35]. Compiling with `defconfig` is fast, because it enables relatively few options, therefore covering very little of the code. Our baseline for a statement-covering configuration file is `make allyesconfig`, which attempts to enable as many configuration options as possible. Although mutual exclusion among configuration options prevents coverage of all code, it still covers the large majority of code (and therefore patches), at the cost of much longer build times. While VAMPYR improves on `allyesconfig`'s coverage, it always builds `allyesconfig` plus additional configuration files [66]. Thus, it always causes even higher build times than `allyesconfig`. We repair `defconfig` by applying krepair to the configuration file to ensure patch coverage, which we expect to achieve the best of both worlds, the high patch coverage of `allyesconfig` and the much faster build times of `defconfig`. Additionally, we evaluate the patch coverage capability of randomly generated configuration files, which is a lightweight method to attempt to increase code coverage. When evaluating random configuration testing, we use Linux's built-in random configuration file generator (`make randconfig`). For evaluating how much change in the configuration file krepair causes, we compare against `defconfig` as well as `allnoconfig`. `allnoconfig` is the Linux kernel's minimal configuration file that disables most configuration options, and we use it as an extreme test case for krepair since it covers few patches.

Metrics collection. To collect metrics for a configuration file on one patch from the sample, we first check out the kernel using the patch's commit ID. We configure and build the kernel using each of the tested configuration files, collecting patch coverage and build time. Patch coverage is evaluated by saving the source code of the patched files after they are configured by the build system, i.e., the preprocessed *.i* files, and checking which lines of the patch have been included or excluded by the build system. We quantify patch coverage as the ratio of changed (added or removed) lines included in the build over the total number of changed lines in the patch.[3] When the patch adds an entirely new file, we consider each line in the file as added by the patch. When the patch removes a file, we consider it as having no lines, since there is no way to build the affected lines of code. Removed lines from an existing file, however, are measured by looking at whether the enclosing `#ifdef` block around the removed line (or the entire file, if there is no `#ifdef`) is included by the configuration file, since that controls the inclusion of the change and does get built. We quantify build time by recording the wall clock time of the build process (`make`) using the UNIX `time` utility.

Parallel builds. `make` supports parallel builds with the `-j` flag, which allows `make` to compile source files in parallel when there are no dependencies between them. Parallel builds do not affect patch coverage or build size, since the same files are compiled with the same configuration file; they only affect build time. For our build time comparisons, we use eight concurrent build threads, since this reflects the power of modern developer laptops, as well as a single thread to record sequential build time. We show the effect on performance of parallel builds in Section 5.3 by comparing the sequential and parallel build times of the kernels in the sample.

Cross-compilation. We run our experiments on a 64-bit x86 machine, but some patched source code can only be built for non-x86 architectures. The patch format does not force the developer to

[3]Non-source files, such as documentation or example source code, are not considered in the total lines, since they are never compiled into the binary.

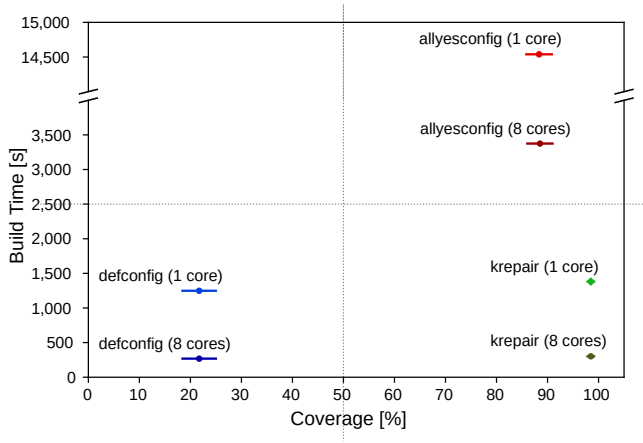

Fig. 4. Patch coverage plotted against build times.

specify for what architecture the code is meant to be built; indeed, one patch may touch code built for multiple architectures. krepair, however, can detect for which architecture(s) a patch is built by exploring the space of configuration constraints for each architecture's Kconfig specification. To build code for other architectures, we perform cross-compilation using the `make.cross`[4] utility, which automates downloading and installing build tools for most other architectures (2 patches are from an architecture that is not supported, so we could not automatically evaluate their patch coverage). We find that 11% of patches in our sample require cross-compilation.

Computing platform. All experiments were run on a server with dual AMD EPYC 7742 64-Core Processors and 512GB of RAM running Ubuntu 22.04.03 LTS. Since this machine allows for high parallelism and our builds are only using either one or eight threads, we parallelized the experiment scripts. All experiment scripts are available in the code repository [6] as well as the artifact archive [77] under `scripts/krepair_evaluation/paper/`.

5.2 Research Questions

We ask the following research questions (RQs) to evaluate krepair:

- RQ1 (Efficient Patch Coverage) Does krepair produce configuration files with high patch coverage and fast build times?
- RQ2 (Performance) How fast is krepair?
- RQ3 (Configuration Preservation) How well does krepair preserve the settings of the repaired configuration file?
- RQ4 (Random Testing) How well does random configuration testing cover patches compared to krepair?
- RQ5 (Build Errors) Can krepair help reveal build errors?

5.3 RQ1: Efficient Patch Coverage

In our experiment, we collect patch coverage and build-time metrics when building each commit in the sample using configuration files made using `make defconfig`, `krepair` to repair `defconfig`, and `make allyesconfig` (`randconfig` will be evaluated in RQ4). Figure 4 compares patch coverage (x-axis, higher is better) to the build time (y-axis, lower is better). Each point is the average of all

[4]https://github.com/fengguang/lkp-tests/blob/master/sbin/make.cross

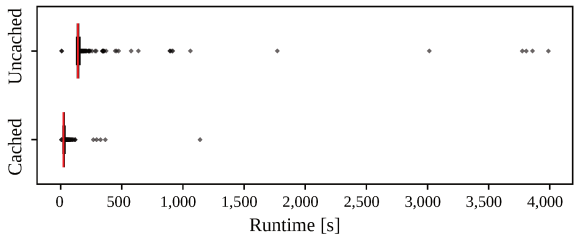

Fig. 5. krepair's cached and uncached runtime.

patch coverage percentages and the average of all build times in seconds across the entire sample of patches, excluding failed builds of which there were 27. The error bars are the 95% confidence interval. For each of the three configuration file generators, we plot both the single-core build time as well as the eight-core parallel build time, resulting in six total points.

Since krepair is intended to preserve fast build times while still covering patched code, we consider it a success if it can simultaneously outperform `defconfig`'s patch coverage and `allyesconfig`'s build time. The results show that repairing `defconfig` with krepair results in much higher patch coverage, about 4.5x more, and even produces higher patch coverage than `allyesconfig`. In contrast, the build times of the repaired `defconfig` are substantially faster, about 10.5x faster, remaining comparable to `defconfig` even after repair. The narrow error bars show that these results from our sample are statistically significant. In short, repairing configuration files with krepair achieves nearly complete patch coverage, while adding little additional build time. Parallel builds reduce build times for all configuration files roughly proportionately.

5.4 RQ2: Performance

We measure krepair's repair runtimes for `defconfig` with and without caching (Section 4.3). Figure 5 is the distribution of times for each patch in the evaluation sample. The Uncached row measures timing when the build-constraints cache is cold and has no prior build constraints cached. The Cached row assumes the build system constraints for each patch have already been cached.

The boxes for the interquartile ranges for both cached and uncached and the lines for the confidence intervals are so narrow, relative the maximum runtime, that they appear to be single a red line on the graph. In other words, the large majority of krepair runtimes take only a few minutes, even without caching, and there are only a small number of outliers that take up to an hour in rare cases. Caching provides a substantial benefit, reducing the maximum runtime by 71.4% to around 19 minutes and less than one minute for 93.8% of runs.

5.5 RQ3: Configuration Preservation

krepair maintains the build time of the original configuration file because it keeps the set of changes to an existing configuration file small when ensuring patch coverage. Therefore, by starting with a fast-building configuration file, such as the default configuration file, the build time is largely preserved to be fast, while patch coverage increases, as shown in Figure 4. We measure how much change krepair incurs to ensure patch coverage to demonstrate its effectiveness at preserving the original configuration file. We define change as the number of configuration options that differ in their setting between two configuration files divided by the total number of configuration options available.

Table 2. Distribution of percent change incurred by krepair.

Comparisons	Min	25th	Median	99th	Max
allnoconfig	0.46%	1.06%	1.48%	2.23%	4.98%
defconfig	0.14%	0.21%	0.27%	1.53%	9.52%

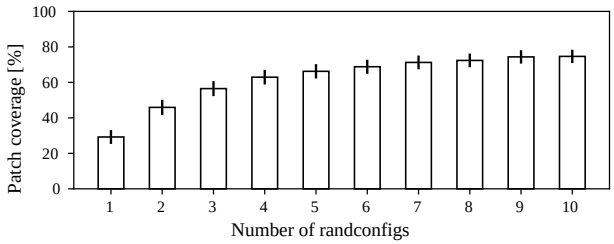

Fig. 6. Aggregate coverage of random configuration files.

In addition to evaluating krepair's repair of `defconfig`, we also evaluate its effectiveness on `allnoconfig`. `allnoconfig` is the Linux kernel's minimal configuration file that disables most configuration options and therefore is an extreme test case for krepair since it covers few patches.

Table 2 is the percentile distribution of change, as a percentage of all configuration options available in the kernel, incurred by krepair when repairing both `defconfig` and `allnoconfig`. For `defconfig`, it changes no more than 10% of the configuration options at the extreme, and no more than 1.53% for 99% of the patches in the sample.

Similarly, krepair finds that only a relatively small amount of change is needed for `allnoconfig`, in spite of it having few options enabled, in order to cover the patches in the sample. In the majority of cases, `allnoconfig` requires more changes than `defconfig`, which is to be expected, given how few options the configuration selects initially. But at the extreme cases, `defconfig` requires more changes; when a configuration file has comparatively more options enabled, there is a chance that the enabled options contradict the dependencies needed for covering the patch, requiring krepair to first disable these options, causing higher total change.

5.6 RQ4: Random Testing

Random configuration file testing is commonly used in continuous configuration testing, so we also evaluate how the patch coverage from multiple random configuration files compares to krepair's coverage. For each patch in our sample, we generate a series of ten random configuration files, measuring their aggregate patch coverage. Figure 6 shows the aggregate patch coverage of 1 to 10 randomly generated configuration files for each patch in the sample. The error bars show the 95% confidence interval of the average for the sample.

Generating a single random configuration file results in low patch coverage, at 29.2%. Increasing the number of configuration files increases coverage, but with diminishing returns; the amount of additional coverage plateaus at 9 random configurations, which in aggregate only cover around 74.4% of patches on average. In contrast, krepair achieves much higher patch coverage, 98.5% on average, and almost always with a single configuration file instead of having nine configuration files requiring more build time.

Table 3. Build errors found when evaluating krepair.

Build Error	Commit ID(s)
Warnings treated as errors	3f977c57, c1318b39, 6ece49c5, c974f755, 5dee8bb8
Linker error	0258cb19, e0905322, 16dd1fbb, dfbdcda2, 661c399a
Implicit function declaration	f9135821, b5054161, ae9fd76f, 4a46e5d2
Incompatible pointer type	800fe5ec, c8992cff
Frame size error	8763e4c1
Undeclared variable	bce84458

5.7 RQ5: Build Errors

In our experiments, we found that some configuration files we generated for patch coverage failed to build. This is unsurprising, since it is infeasible for developers to build all variants of the kernel. While `defconfig` and `allyesconfig` are frequently tested and typically do not trigger build errors in released code (code given a version number), small variations in a configuration file can expose new bugs.

We found 18 build errors due to several bugs. Table 3 lists the build errors found, with the commit ID(s) through which they were found. These bugs were not introduced by the corresponding commits, but were present at the checkout of the commit. Ten were due to missing symbol declarations (linker errors, implicit function declarations, undeclared variables). Missing declarations occur in highly-configurable software when the declaration of a symbol is disabled by one configuration option and the use of the symbol is enabled by another. Five build errors were due to `-Werror` being enabled by the configuration file, causing compiler warnings (which by default do not halt compilation) to trigger a compiler error. Two commits failed due to pointer type mismatches, and one due to a display mode subsystem error: "the frame size of 2112 bytes is larger than 2048". We patched one of two build errors still replicable in the recent v6.1-rc8 kernel. This patch has been accepted for inclusion in mainline Linux, while we plan to patch the other. The rest of the bugs were no longer in the most recent kernel.

Since we build non-x86 patches on x86 hardware, we could not cross-compile some of the patches due to limitations of our cross-compilation tooling, `make.cross`. The `make.cross` script does not support the newly-added `loongarch` architecture and some cross-compilers had incompatibility, for instance, reporting unexpected assembly opcodes. These cross-compilation problems prevented us from building seven patches: 8c4d1647, 0b452520, 7eafa6ee, 44c14509, 6982dba1, f62b7626, 54cfa910. krepair determined the parisc 32-bit architecture for two commits, 53d862fa and db2b0d76, while the configuration files instead required the parisc 64-bit cross-compiler, which is not available with `make.cross`.

6 THREATS TO VALIDITY

Internal validity. Since krepair relies on existing constraint collection tools [31, 33, 53], any limitations of these tools limit krepair. Specifically, these tools only collect constraints from the build system, while other sources of constraints are not supported, such as run-time uses of configuration options, i.e., with C conditionals instead of #ifdef. Additionally, header file inclusion constraints are also not available from these tools, though future work on constraint collection could yield analyses that discover all possible ways a header file is included across the entire kernel source. Even without the above limitations, 100% patch coverage may not necessarily be possible in all cases, as some patches change dead code in `#ifdef 0` blocks, which can never be included in any build.

External validity. We evaluate krepair on only one software system, the Linux kernel source code, albeit one of the largest and most highly-configurable open-source software products. While our implementation is targeted to the Linux build system, this build system is also used by numerous systems and embedded open-source projects (BusyBox, coreboot, zephyr, etc.). krepair's algorithm, however, is independent of any particular build system, since it operates on any configuration constraints extracted from the software product. krepair focuses on the problem of patch coverage to enable more efficient continuous integration, since current approaches cannot even guarantee that patched lines are built. Testing all the effects of a patch, however, goes beyond just line coverage; succeeding in compiling does not guarantee a test suite will execute the code without additional analysis. Patch coverage is the first step to any kernel testing, so we are exploring future work on combining our repair approach with kernel fuzz-testing [35], change impact analysis [55], configuration interaction testing [76], and other testing strategies [28].

7 RELATED WORK

To the best of our knowledge krepair is the first technique to repair Linux configuration files for patch coverage. We highlight work related to krepair and that addresses related problems in the domain of configurable software.

Configuration coverage. JMake [45] is a previous attempt to find a configuration that covers a patch. However, it tries only a fixed set of standard configurations. JMake also introduces a mutation-based approach to determining if a line of code is covered. Acher et al. [9] explore the effect of configurations on compiled Linux kernel sizes, and compare machine learning approaches for predicting compiled size from configurations. They also explore small Linux builds and their use cases. Tartler et al. [67] introduce a metric for how much of the source code is covered by a configuration. Motivated by the results obtained for this metric, Tartler et al. [66] created VAMPYR, a statement-maximizing approach discussed in Section 2. Note that VAMPYR is an older tool and no longer maintained. It only supports up to around Linux 3.2 (released in 2012).

Configuration constraint finding. krepair takes inspiration from prior work on collecting constraints from Linux build-system code to get patch-covering constraints during repair. Several prior works extract constraints from Kconfig specifications by translating Kconfig language constructs into logical formulas or feature models [24, 44, 53, 57, 58]. Kbuild Makefile analysis collects logical constraints using both static and dynamic program analyses [13, 31, 51]. Several C preprocessor static code analyzers model configuration constraints in logic [30, 33, 40, 60, 72], albeit for parsing, type-checking, refactoring, bug-finding, etc., rather than constraint extraction. Prior work on localizing configuration constraints per-line aggregates constraints from multiple sources, including Kconfig, Kbuild, and the C preprocessor [34, 41], although it does not scale to the Linux build system. Collecting line constraints is not enough to create a valid Linux kernel configuration file, due to the need to additionally incorporate basic system functionality and the possibility of conflicting constraints. While krepair is the first tool we know of to automatically achieve patch coverage, there are applications of configuration constraints in prior work to other software engineering problems, including attack surface reduction [42, 43], dead code elimination [68], statistical analysis of build errors [8], configuration tracing [29] and configuration specification bug-finding [53].

Analyses for other configuration systems. The Puppet deployment configuration language has formal verification by Shambaugh et al. [56], automated repair by Weiss et al. [70], and a formal model of the system call trace by Sotiropoulos et al. [59]. Formal models are also used for system configuration script and resource usage by Hanappi et al. to test if a system is recoverable [37], as well as by Bouchet et al. [14] to check for public access to Amazon S3 instances. Horton et al. [38] infer dependencies from Python code snippets to produce Docker specifications. Sun et al. [61] introduce ctests to detect potential system failures from configuration changes. Cheng

et al. do configuration test case prioritization [16]. Tamrawi et al. [65] introduce SYMake, which performs static analysis of Makefiles to detect errors like cyclic dependencies and can aid in refactoring. MAKAO, by Adams et al. [10], can be used to create graphs of Makefile dependencies for visualization. Zhang and Ernst explore retaining system behavior after changes [79].

Random sampling for configuration testing. While random configuration testing is difficult to scale to the Linux build system [50], sophisticated testing approaches for smaller systems include genetic algorithms [36], pair-wise feature selection [64], and combinatorial interaction testing [18, 52, 75].

Tracking evolution of Linux patches. krepair's evaluation looks at a sample of patches over time. Prior work has also measured how the configuration system evolves over time, in particular how they relate to code size [46], what patterns are in the mapping between options and implementation [54], how configuration options change over time [25], how patches affect configuration specifications [26], and how changes of Kconfig impact source code [80].

Fixing configuration errors. A related but distinct line of work addresses the problem of fixing configuration errors [69, 71, 73], such as those that appear after code evolution. In using the term "repair" in our work on krepair, we are referring to automatically modifying a valid configuration file to remedy its lack of patch coverage. But we do not address the problem of fixing erroneous configuration files.

8 CONCLUSION

We have shown how krepair achieves much higher coverage of patches in kernel builds via automated repair of configuration files. Its algorithm's design and implementation balance the expense of satisfiability with tool performance to achieve patch coverage comparable to maximal configuration files while preserving most configuration options settings from the repaired configuration file. krepair keeps build times fast while retaining patch coverage, potentially reducing the energy costs of configuration testing which relies heavily on building many randomly-generated configuration files. Our evaluation shows that krepair achieves 4.5x more patch coverage than default configuration files with 10.5x less build time than maximal configuration files on a statistically-significant sample of Linux kernel patches. For future work, we plan to extend krepair to other problems, such as fuzz-testing, change impact analysis, configuration bisection, and other testing and analyses for highly-configurable software.

9 DATA AVAILABILITY

The krepair tool has been released as free-and-open-source software as part of the kmax tool suite [6] and has also been archived on Zenodo [77]. The scripts to run experiments and the resulting data has been archived on Zenodo [78].

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their valuable comments. This work is supported in part by the National Science Foundation under grant CCF-1941816.

REFERENCES

[1] 2018. Busybox website. https://busybox.net/.
[2] 2020. syzkaller commit 67fa1f59b87f "executor: add support for USB fuzzing on NetBSD". https://github.com/google/syzkaller/commit/67fa1f59b87fed7268b465f7e9540a590a250c65, Last accessed May 4, 2022.
[3] 2020. syzkaller commit 80a0690249dc "dashboard/config: regenerate all configs". https://github.com/google/syzkaller/commit/80a0690249dc4dbbbed95ba197192b99c73694c5, Last accessed May 4, 2022.
[4] 2021. Mainline Linux Git Repository. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.
[5] 2021. whatthepatch. https://pypi.org/project/whatthepatch/.
[6] 2024. https://github.com/paulgazz/kmax. Accessed: 2024-02-06.

[7] Top 500. 2020. Sublist Generator. https://www.top500.org/statistics/sublist/.

[8] 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

[9] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. 2019. *Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes.* Research Report. Inria Rennes - Bretagne Atlantique. https://hal.inria.fr/hal-02314830

[10] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France.* 114–123. https://doi.org/10.1109/ICSM.2007.4362624

[11] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[12] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).

[13] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines.. In *SPLC*. 498–499.

[14] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Dan Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. 2020. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 281–291.

[15] Jesper Dangaard Brouer. 2016. Kernel Software Variability: From a kernel developer's perspective: commonly known as #ifdef challenges. https://people.netfilter.org/hawk/presentations/ifdef2016/ifdef_FOSD2016.pdf Keynote Talk, Feature-Oriented Software Development (FOSD).

[16] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 452–465. https://doi.org/10.1145/3460319.3464810

[17] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. 1996. The combinatorial design approach to automatic test generation. *IEEE Software* 13, 5 (Sept. 1996), 83–88. https://doi.org/10.1109/52.536462

[18] Myra Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *Software Engineering, IEEE Transactions on* 34 (09 2008), 633–650. https://doi.org/10.1109/TSE.2008.50

[19] Jonathan Corbet. 2020. Some 5.5 kernel development statistics. https://lwn.net/Articles/810639/.

[20] Jonathan Corbet. 2021. Some 5.12 development statistics. https://lwn.net/Articles/853039/.

[21] Coreboot. [n. d.]. https://www.coreboot.org.

[22] Intel Corporation. 2021. 0-Day Test Service. https://01.org/lkp/documentation/0-day-test-service.

[23] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[24] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikshat, and Daniel Lohmann. 2012. Understanding Linux Feature Distribution. In *Proceedings of the 2012 Workshop on Modularity in Systems Software* (Potsdam, Germany) *(MISS '12)*. ACM, New York, NY, USA, 15–20. https://doi.org/10.1145/2162024.2162030

[25] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. Analysing the Linux kernel feature model changes using FMDiff. *Software & Systems Modeling* 16, 1 (2017), 55–76.

[26] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering* 23, 2 (2018), 905–952.

[27] Eclipse Foundation. 2018. IoT Developer Survey Results. https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2018.pdf. Accessed: 2020-06-10.

[28] The Linux Foundation. 2022. KernelCI. https://foundation.kernelci.org/, Last accessed 05/04/2022.

[29] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. 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)*. IEEE, 91–100.

[30] Alejandra Garrido and Ralph Johnson. 2005. Analyzing multiple configurations of a C program. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 379–388.

[31] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 279–290. https://doi.org/10.1145/3106237.3106283

[32] Paul Gazzillo. 2020. Inferring and Securing Software Configurations Using Automated Reasoning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1517–1520. https://doi.org/10.1145/3368089.3417041

[33] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 323–334. https://doi.org/10.1145/2254064.2254103

[34] Paul Gazzillo, Ugur Koc, ThanhVu Nguyen, and Shiyi Wei. 2018. Localizing configurations in highly-configurable systems. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 269–273.

[35] Google. 2020. syzkaller. https://github.com/google/syzkaller/.

[36] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software* 84 (12 2011), 2208–2221. https://doi.org/10.1016/j.jss.2011.06.026

[37] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 328–343.

[38] Eric Horton and Chris Parnin. 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 328–338.

[39] IEEE. 2012. IEEE Standard for Configuration Management in Systems and Software Engineering.

[40] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 805–824. https://doi.org/10.1145/2048066.2048128

[41] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. 2018. PClocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 284–288. https://doi.org/10.1145/3233027.3236399

[42] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1, Article 03 (may 2020), 27 pages. https://doi.org/10.1145/3379469

[43] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring.. In *NDSS*.

[44] Christian Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. arXiv:1706.09357 [cs.SE]

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

[46] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux kernel variability model. In *International Conference on Software Product Lines*. Springer, 136–150.

[47] Inês Lynce and João Marques-silva. 2004. On computing minimum unsatisfiable cores. In *In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*. 305–310.

[48] Alan Maguire. 2021. A Zoological guide to kernel data structures. https://blogs.oracle.com/linux/post/a-zoological-guide-to-kernel-data-structures.

[49] GNU Manual. 2022. GNU Diffutils: Unified Format. https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.

[50] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793

[51] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 107–116.

[52] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. T-Wise Coverage by Uniform Sampling. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 84–87. https://doi.org/10.1145/3336294.3342359

[53] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding broken Linux configuration specifications by statically analyzing the Kconfig language. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for

Computing Machinery, 893–905.
[54] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of variability models and related software artifacts. *Empirical Software Engineering* 21, 4 (2016), 1744–1793.
[55] Barbara G. Ryder and Frank Tip. 2001. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01)*. Association for Computing Machinery, New York, NY, USA, 46–53. https://doi.org/10.1145/379605.379661
[56] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A configuration verification tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 416–430. https://doi.org/10.1145/2908080.2908083
[57] Steven She. 2013. Feature model synthesis. (2013).
[58] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 33–42.
[59] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 26–37.
[60] Diomidis Spinellis. 2010. CScout: A refactoring browser for C. *Science of Computer Programming* 75, 4 (2010), 216–231.
[61] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 735–751. https://www.usenix.org/conference/osdi20/presentation/sun
[62] W3Techs World Wide Web Technology Surveys. 2019. Usage statistics of Unix for websites. https://w3techs.com/technologies/details/os-unix/all/all. Accessed: 2020-06-10.
[63] syzbotreport 2021. syzbot report "memory leak in dvb_create_media_graph". https://syzkaller.appspot.com/bug?extid=7f09440acc069a0d38ac, Last accessed May 4, 2022.
[64] Kuo-Chung Tai and Yu Lei. 2002. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 28, 1 (Jan. 2002), 109–111. https://doi.org/10.1109/32.979992
[65] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. SYMake: A build code analysis and refactoring tool for makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 366–369. https://doi.org/10.1145/2351676.2351749
[66] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 421–432.
[67] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration coverage in the analysis of large-scale system software. In *Workshop on Programming Languages and Operating Systems, PLOS@SOSP*. ACM, 2:1–2:5.
[68] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the 6th European Conference on Computer Systems*. 47–60. http://dx.doi.org/10.1145/1966445.1966451
[69] Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang. 2013. SmartFixer: fixing software configurations based on dynamic priorities. In *Proceedings of the 17th International Software Product Line Conference* (Tokyo, Japan) *(SPLC '13)*. Association for Computing Machinery, New York, NY, USA, 82–90. https://doi.org/10.1145/2491627.2491640
[70] Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: Interactive system configuration repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 625–636.
[71] J. White, D.C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz–Cortés. 2008. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *2008 12th International Software Product Line Conference*. 225–234. https://doi.org/10.1109/SPLC.2008.16
[72] Norman Wilde and Michael C Scully. 1995. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice* 7, 1 (1995), 49–62.
[73] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. 2012. Generating range fixes for software configuration. In *2012 34th International Conference on Software Engineering (ICSE)*. 58–68. https://doi.org/10.1109/ICSE.2012.6227206
[74] Karim Yaghmour. 2003. *Building Embedded Linux Systems*. O'Reilly Media, Inc.
[75] C. Yilmaz, S. Fouché, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc. 2014. Moving forward with combinatorial interaction testing. *Computer* 47, 2 (Feb. 2014), 37–45. https://doi.org/10.1109/MC.2013.408
[76] X. Yuan, M. B. Cohen, and A. M. Memon. 2011. GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering* 37, 4 (July 2011), 559–574.

[77] Necip Fazıl Yıldıran, Jeho Oh, Julia Lawall, and Paul Gazzillo. 2024. *Artifact from "Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times"*. https://doi.org/10.5281/zenodo.10626343
[78] Necip Fazıl Yıldıran, Jeho Oh, Julia Lawall, and Paul Gazzillo. 2024. *Experimental data for "Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times"*. https://doi.org/10.5281/zenodo.10626233
[79] Sai Zhang and Michael D. Ernst. 2014. Which configuration option should I change?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 152–163. https://doi.org/10.1145/2568225.2568251
[80] Andreas Ziegler, Valentin Rothberg, and Daniel Lohmann. 2016. Analyzing the impact of feature changes in Linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. 25–32.

Received 2023-09-28; accepted 2024-01-23