JMake: Dependable Compilation for Kernel Janitors

Julia Lawall, Gilles Muller

JMake: Dependable Compilation for Kernel Janitors

Julia Lawall
Sorbonne Universités/Inria/UPMC/LIP6
Email: Julia.Lawall@lip6.fr

Gilles Muller
Sorbonne Universités/Inria/UPMC/LIP6
Email: Gilles.Muller@lip6.fr

***Abstract*—The Linux kernel is highly configurable, and thus, in principle, any line of code can be included or excluded from the compiled kernel based on configuration operations. Configurability complicates the task of a kernel janitor, who cleans up faults across the code base. A janitor may not be familiar with the configuration options that trigger compilation of a particular code line, leading him to believe that a fix has been compile-checked when this is not the case. We propose JMake, a mutation-based tool for signaling changed lines that are not subjected to the compiler. JMake shows that for most of the 12,000 file-modifying commits between Linux v4.3 and v4.4 the configuration chosen by the kernel allyesconfig option is sufficient, once the janitor chooses the correct architecture. For most commits, this check requires only 30 seconds or less. We then characterize the situations in which changed code is not subjected to compilation in practice.**

I. Introduction

A *janitor* is a software developer who focuses on finding and fixing faults, coding style violations, and out-of-date code across a code base. A janitor works breadth-first, addressing issues wherever they occur, in contrast to a *maintainer*, who works depth-first on the issues affecting in a single subsystem. While a maintainer may have deep technical knowledge about the functioning of a particular system service or device, this is not the goal of a janitor, who instead contributes an awareness of the latest APIs, common pitfalls, and the latest coding style improvements.

Janitors play an important role in the development of the Linux kernel. The Linux kernel is a very large open-source system, currently amounting to over 13 million lines of C code, and having hundreds of active developers, with varying levels of expertise. It is widely used, in environments ranging from embedded systems, to servers, to supercomputers, making its dependability critical. Given the large size of the Linux kernel and the wide range of expertise of its developers, faults, coding-style violations, and out-of-date code seem inevitable, as has been substantiated by various studies [1], [2]. It is thus essential to support the detection and cleanup of these issues, and thus the work that Linux kernel janitors do.

A challenge for a janitor in working on the Linux kernel is how to check the correctness of his changes. Indeed, much of the Linux kernel targets specialized services, such as devices and file systems, that a janitor may not have locally available. It has been estimated that only 1.6 million lines of Linux kernel code are actually executed on a typical Linux-based laptop.[1] Furthermore, even for locally available services, many lines of code handle rare conditions. Thus, achieving high test coverage is difficult for a developer who is not an expert in the specific code. Accordingly, evidence in patch submissions suggests that janitors often only check that the file containing their proposed changes successfully compiles.

Even checking that all of the lines affected by a proposed change to the Linux kernel have been subjected to the compiler is challenging, because the Linux kernel is highly configurable. For example, the Linux kernel v4.4, released in January 2016, defines around 15,000 configuration variables, that work together to control what parts of the kernel source code are compiled for inclusion in a kernel image. Files may fail to compile for standard configurations when they rely on properties of the compile-time environment such as include paths that are only correctly put in place for specific configurations; the resulting compiler errors can discourage the janitor, as they include no information about how to solve the problem. Even worse, some lines within a file may be included only by specific configurations, giving the janitor a false sense of security when the file compiles but some of the changed lines have not actually been subjected to the compiler. While Intel has put in place a "0-day build testing service" [3] that compiles all patches for a number of randomly selected configurations and reports back to the patch submitter on any issue, these tests are also not exhaustive [4] and the timing of the feedback is not under the control of the patch submitter.

In this paper, we propose an approach to help the janitor find an appropriate configuration for compiling the file in which he has made changes, and to provide immediate feedback to the janitor on whether this configuration actually causes the changed lines to be compiled. Our approach is based on a combination of mutations of the lines affected by a given patch, heuristics for identifying the architecture targeted by a particular file, and testing of compilations for these different architectures. We have implemented this approach in a prototype tool JMake that provides a practical tool for a janitor to identify parts of his changes that are not subjected to the compiler. JMake also provides a means of assessing the overall degree of difficulty introduced by the configurability of the Linux kernel, by identifying the prevalence of patches that are not thoroughly compile-tested by standard configurations in the Linux kernel as a whole.

The contributions of this paper are as follows:

- We provide an approach that certifies to the janitor that a successful compilation implies successful compilation of all of the changed lines.

[1] Greg Kroah Hartman, informal communication.

This paper appeared in 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 357-366.

- We identify the problem of changed lines that are not subjected to the compiler in the compile testing of highly configurable systems.
- We propose an approach to selecting promising configurations and detecting changed lines that are not subjected to compilation by a given compiler invocation.
- We provide a prototype of JMake based on the design of our approach.
- Using JMake and the complete set of patches applied to the Linux kernel between versions v4.3 (11.2015) and v4.4 (01.2016), we confirm the possibility of changed lines in Linux kernel patches that are not subjected to the compiler when using standard comprehensive configurations.
- We propose a characterization of a janitor in the context of Linux kernel development, and study how the problem of changed lines that are not subjected to the compiler affects Linux kernel janitor patches.

The rest of this paper is organized as follows. Section II gives background about Linux kernel configuration and about the Linux kernel development process. Section III presents JMake. Section IV presents our proposed characterization of a janitor. Section V presents the design and results of our evaluation. Section VI presents related work and Section VII concludes.

II. Background

We begin with some background, on the Linux kernel, on the Linux kernel configuration mechanism, and on the notion of a patch.

A. The Linux kernel

The Linux kernel is an open source operating system kernel that has been in active development since 1994. A new release of the Linux kernel appears roughly every 3 months. We take as our reference version Linux v4.4, released in January 2016.

The Linux kernel code base is organized as a number of subdirectories containing code for various kinds of services. Among these is `arch`, containing subdirectories for each supported architecture. As of Linux 4.4, the `arch` directory contains 32 subdirectories, some of which support multiple architecture variants.

In this paper, we are primarily concerned with the process of compiling the Linux kernel source code. A number of Linux kernel files, even those outside the `arch` subdirectory, are specific to one or more architectures, due to dependencies on architecture-specific libraries and header (`.h`) files. Compiling files targeting architectures other than that of the host machine requires cross-compilation tools. The script `make.cross`[2] provides a transparent interface to many of the tools available. This script currently supports 34 architectures, however, we were only able to make compilation work for 24 of them.[3]

The Linux kernel Makefile provides a variety of targets that make it possible to compile the entire kernel (`make`) or to compile a single file (`make file.o`). Another option provided is to apply the preprocessor to a single file (`make file.i`). JMake primarily relies on the latter two options.

B. Linux kernel configuration

The Linux kernel build system is based on Kbuild, which provides a highly sophisticated means of defining configuration variables and managing their values. The details of Kbuild are presented elsewhere [5] and are not central to this paper. The Linux kernel provides the make target `allyesconfig` that attempts to set as many configuration variables as possible, as long as doing so does not conflict with the chosen architecture or any of the other chosen options. An alternative is `allmodconfig`, which compiles as many subsystems as dynamically loaded kernel modules as possible. The Linux kernel networking documentation, for example, explicitly suggests that all submitted changes should be first tested with the `allyesconfig` and `allmodconfig` configurations.[4] JMake currently focuses on `allyesconfig`. The various architectures supported by the Linux kernel also provide a selection of sample configurations in the `configs` subdirectory of the architecture support code. JMake also takes advantage of these configurations.

C. Patches

In the context of Linux kernel development, changes are typically reasoned about in terms of *patches*. A patch consists of a sequence of extracts of the source code, in which some lines are annotated with − indicating that they are to be removed, some lines are annotated with + indicating that they are to be added, and, optionally, some lines are unannotated, to provide some extra information about the context in which the change should occur.

A patch can be produced using the Unix tool `diff`, and applied to a copy of the original source code using the Unix tool `patch`. Version control systems, such as `git` used by the Linux kernel, typically provide support for visualizing changes in the form of a patch. When using git, for a particular commit identifier *id*, the command `git show` *id* produces a patch describing the changes made in that commit. JMake identifies changed lines by analyzing patches, and produces patches to mutate the source code.

III. Approach

The basic idea of our approach is to mutate the source code by inserting unique tokens into the code at the change sites, then to compile the code, and finally to check that all of the unique tokens are found in the compiled image. To carry

[2]https://git.kernel.org/cgit/linux/kernel/git/wfg/lkp-tests.git/plain/sbin/make.cross

[3]The architectures i386, x86_64, alpha, arm, avr32, blackfin, cris, ia64, m32r, m68k, microblaze, mips, mn10300, openrisc, parisc, powerpc, s390, sh, sparc, sparc64, tile, tilegx, um, and xtensa worked. The architectures arm64, c6x, frv, h8300, hexagon, score, sh64, sparc32, tilepro, and unicore32 failed for various reasons.

[4]Documentation/networking/netdev-FAQ.txt

```
1 // .c file, from line 49
2 #define DAS16CS_AI_MUX_HI_CHAN(x) (((x) & 0xf) << 4)
3 #define DAS16CS_AI_MUX_LO_CHAN(x) (((x) & 0xf) << 0)
4 #define DAS16CS_AI_MUX_SINGLE_CHAN(x) \
5    (DAS16CS_AI_MUX_HI_CHAN(x) | \
6     DAS16CS_AI_MUX_LO_CHAN(x))
7 ...
8 static int das16cs_ai_rinsn
9         (struct comedi_device *dev, // line 107
10         struct comedi_subdevice *s,
11         struct comedi_insn *insn, unsigned int *data)
12 {
13         struct das16cs_private *devpriv = dev->private;
14         int chan = CR_CHAN(insn->chanspec);
15         int range = CR_RANGE(insn->chanspec);
16         int aref = CR_AREF(insn->chanspec);
17         int ret;
18         int i;
19
20         outw(DAS16CS_AI_MUX_SINGLE_CHAN(chan),
21              dev->iobase + DAS16CS_AI_MUX_REG);
22         ...
23 }
24
25 // .i file
26 # 43 "drivers/staging/comedi/drivers/cb_das16_cs.c" 2
27 # 58 "drivers/staging/comedi/drivers/cb_das16_cs.c"
28 ...
29 static int das16cs_ai_rinsn(struct comedi_device *dev,
30        struct comedi_subdevice *s,
31        struct comedi_insn *insn, unsigned int *data)
32 {
33  struct das16cs_private *devpriv = dev->private;
34  int chan = ((insn->chanspec)&0xffff);
35  int range = (((insn->chanspec)>>16)&0xff);
36  int aref = (((insn->chanspec)>>24)&0x03);
37  int ret;
38  int i;
39
40  outw(((((chan) & 0xf) << 4) | (((chan) & 0xf) << 0)),
41       dev->iobase + 0x02);
42  ...
43 }
```

Fig. 1. Extract of `drivers/staging/comedi/drivers/cb_das16_cs.c` after applying Linux kernel commit 95ea3e760ef8

out this approach, we must address a number of issues, as described below.

A. Choosing relevant artifacts

Our goal is to determine which lines of the source code are subjected to the compiler. For simplicity and portability, we want to avoid changing the compiler. Accordingly, we are limited to modifying its input and extracting information from its output. For the input, we can either simply note the line number and contents of the lines that we want to track, or modify these lines in some way. For the output, we can analyze any messages generated during the compilation process as well as the contents of the various possible generated files: `.i` files containing the results of preprocessing, `.s` files containing the corresponding assembly language code, `.lst` files containing the assembly language code interspersed with the corresponding C code, and `.o` files containing the object code. We show small extracts of a `.c` files and its corresponding `.i` file in Figure 1. Note how the macros defined in the first 5 lines of the `.c` file extract (lines 2-6) are absent in the `.i` file extract (line 26-27), but they are inlined into their usage sites in the subsequent `.i` code (line 40).

The strategy we take is primarily determined by the need to track the compilation of code found in macro definitions. Indeed, lines within a macro definition are processed by the compiler at the place where the macro is used, and not where it is defined, and the original line numbers of macros are not preserved in the `.i`, `.s`, and `.lst` files. Our approach is to mutate the source code with some unique string that can be recognized in the generated files wherever it occurs. If the mutation results in invalid source code, such as an undefined variable, then we may be able to observe the effects of the mutation in the compiler error messages. The reliability of this approach, however, depends on the compiler's error reporting policy. For example, we have found when using gcc 4.8.4 that an unbound variable in a macro definition will trigger an error message that mentions the macro definition itself, but an invalid character in the macro definition only triggers an error message on the place where the macro is used, which is not sufficient to determine which line of source code has been compiled. Reliably choosing a variable that is considered to be unbound at any given point in the kernel is difficult, particularly given that included files can define macros that construct new identifiers. Therefore, we choose to include an invalid character in our mutations, and give up on the inherently unreliable idea of using compiler error messages to detect the compilation of mutated lines of code.

Our reliance on mutations that contain at least one invalid character implies that JMake will not be able to produce a `.s`, `.lst`, or `.o` file from a mutated file, as all of these are only generated for files that pass all the verifications of the compiler front end. We turn instead to the `.i` file, which is produced by only interpreting the preprocessor directives. The `.i` file has the advantage of showing the code that will be subjected to the compiler, without starting the compilation process. Mutations involving invalid characters are reproduced as is in a `.i` file when the mutation is outside of a macro definition, and are propagated unchanged into the macro usage sites otherwise. Producing a `.i` file for a given configuration, however, does not imply that the configuration enables successful compilation. We thus use the `.i` file to determine whether each changed line is subjected to the compiler, and then try to produce a `.o` file from the original, unmutated code to ensure that it is possible to compile the file containing those changed lines.

Concretely, our approach mutates the code by inserting strings of the form `¤"type:file:line"¤` at the change sites. The invalid ¤ characters ensure that the mutation is different from any valid C code. The type gives some information about the kind of code change involved. The file and line indicate the position of the changed line in the source code. Finally, we put these pieces within a string to protect them from modification by the preprocessor.

B. Positioning of mutations

We next consider where mutations should be placed in the code. Our goal is to place the mutations such that the presence of all of the mutations in the `.i` file indicates that all of the

changed lines are subjected to the compiler. At the same time, we try to minimize the number of mutations, to minimize the amount of code that has to be studied when JMake reports that some changed lines are not compiled. We distinguish three types of changed lines: 1) lines within a comment, 2) lines within a macro definition, and 3) other lines. Lines within a comment are never processed by the compiler, and are thus not relevant to JMake. We thus focus on the other two cases.

For changes in a macro, we assume that the macro does not contain any conditional compilation directives and thus that it is sufficient to insert one mutation in any changed macro definition. If the first change within a macro definition is on the `#define` line, the mutation is placed at the end of the line so that it is considered to be part of the macro definition. If the `#define` line ends with a continuation character `\`, the mutation is placed just before the continuation character. On the other hand, if the first changed line in a macro definition is on some line other than the first one, then a new line with only a mutation and a continuation character is added before the first modified one. Figure 2 shows some macro definitions with their associated mutations.

For code that is not within a macro definition, we only need one mutation since the beginning of the file, or since the most recent conditional compilation directive, *i.e.*, any line beginning with `#if` (including `#ifdef` and `#ifndef`), `#else`, or `#elif`. Normally, the mutation is added to a new line before the changed one. If, however, the changed line begins in the middle of a comment that ends in the current line, then the mutation is placed after the end of the comment. Figure 3 shows some non-macro code with the associated mutations.

Mutations and all other processing are only performed on the code generated by the patch. A patch may add or remove code as well as change it. In the case of a hunk that only adds code, the changed lines are considered to be the added lines. In the case of a hunk that only removes code, the changed line is considered to be the first line remaining after the removed code, or the end of the file.

C. Selection of possible architectures

We assume that a file found in a subdirectory of `arch` can be compiled by the cross-compiler for that architecture. For other files, we guess the possible architectures using a collection of heuristics. The first guess is a simple `make`, with no cross compilation. Indeed, the Linux kernel provides the option `CONFIG_COMPILE_TEST`, introduced in Linux 3.11, that has the goal of allowing compilation of device drivers even when the device driver cannot be run on the current hardware.

Other hints are provided by the configuration variables associated with the compilation of a given file in the associated Makefile. If such a configuration variable is also mentioned somewhere in a subdirectory of `arch`, then the file may be compilable for the corresponding architecture. In this case, we try the `allyesconfig` make target for the given architecture to create an appropriate configuration. Configuration variables are taken from Makefile lines that mention the `.o` file corresponding to the C file to compile, recursively from the lines containing labels that are initialized to contain such a `.o` file, or, if the previous heuristics do not select any configuration variables, then any configuration variable mentioned in the Makefile. If such a configuration variable appears in one or more files in the `configs` subdirectory of a subdirectory of `arch`, then JMake additionally uses one such configuration file chosen at random.

D. Processing of `.c` files

Processing of a `.c` file involves 1) applying the mutation patch to the `.c` file, 2) compiling the mutated `.c` file into the corresponding `.i` file, 3) searching for the various mutations in the `.i` file, and 4) if all are present, compiling the original, unmutated `.c` file into the corresponding `.o` (object) file. This process is repeated for each of the selected architectures, until all of the mutations have been subjected to the compiler for at least one configuration where compilation succeeds or all of the selected architectures have been tested. In the former case, representing success, JMake reports on the architectures for which compilation was successful and that reduced the number of lines remaining to be subjected to the compiler. In case of failure, JMake returns the list of mutations that were not found, or an indication of the other possible errors, such as no Makefile found, an unsupported architecture required, or a failure in making the `.i` or `.o` file.

The Linux kernel compilation environment is complex, and thus, for example, the Linux kernel v4.4 Makefile performs many tens of set up operations (*e.g.*, over 80 for x86 and over 60 for arm) when first called on a new configuration, before processing the targeted files. The Makefile furthermore performs a small number of extra checks on each subsequent invocation for the same configuration. Thus, we can reduce the overall running time by running make on multiple files at once. Typically, a patch will affect multiple `.c` files, and thus for each selected architecture, JMake compiles together all of the `.c` files from a given patch that are relevant for that architecture, in groups limited to a size that the user expects will avoid overflowing the space resources of the machine.

A given patch may modify both `.c` files and `.h` files, and the `.h` file changes may be designed to support the code in the patched `.c` files. Thus, while processing the `.c` files we also mutate the `.h` files and keep track of the `.h` file mutations that show up in the `.i` files. If all of the mutations in a `.h` file appear in the `.i` files of the changed `.c` files, for successfully compiled configurations, then the processing of the `.h` file is also considered a success, and the `.h` file is not subjected to the further processing described in the next section.

E. Processing of `.h` files

While our treatment of `.c` files can detect some mutants in `.h` files, some changes in `.h` files may not be relevant to the `.c` files in the patch, and some patches may contain no `.c` files. In this case, because it is not possible to compile a `.h` file directly, we need to select `.c` files that may reference the

```
#define DAS16CS_AI_MUX_HI_CHAN(x)      (((x) & 0xf) << 4)¤"define:drivers/staging/comedi/drivers/cb_das16_cs.c:49"¤
#define DAS16CS_AI_MUX_LO_CHAN(x)      (((x) & 0xf) << 0)¤"define:drivers/staging/comedi/drivers/cb_das16_cs.c:50"¤
#define DAS16CS_AI_MUX_SINGLE_CHAN(x) (DAS16CS_AI_MUX_HI_CHAN(x) | ¤"define:drivers/staging/comedi/drivers/cb_das16_cs.c:51"¤\
                                       DAS16CS_AI_MUX_LO_CHAN(x))
```

Fig. 2. Mutations in macros of `drivers/staging/comedi/drivers/cb_das16_cs.c`

```
static int das16cs_ai_rinsn(struct comedi_device *dev,
              struct comedi_subdevice *s,
              struct comedi_insn *insn, unsigned int *data)
{
        struct das16cs_private *devpriv = dev->private;
        int chan = CR_CHAN(insn->chanspec);
        int range = CR_RANGE(insn->chanspec);
        int aref = CR_AREF(insn->chanspec);
        int ret;
        int i;

¤"noncomment:.../staging/comedi/drivers/cb_das16_cs.c:118"¤
        outw(DAS16CS_AI_MUX_SINGLE_CHAN(chan),
             dev->iobase + DAS16CS_AI_MUX_REG);
        ...
}
```

Fig. 3. Mutation in non-macro code of `drivers/staging/comedi/-drivers/cb_das16_cs.c`

changed elements. One likely set of files is those that include the given `.h` file directly. But one header file may include another, and detection of the complete set of files included by a `.c` file would require an expensive recursive analysis. Instead, as header files are commonly used to define macros, we extend the mutation process to collect the set of names of macros that are affected by changes. We then use these macro names as hints, taking also all `.c` files that refer to these names.[5] Finally, if a `.h` file is in a subdirectory of `arch`, it is expected to only be relevant to `.c` files in the same `arch` subdirectory or to non-`arch` files. Our approach orders the resulting `.c` files such that those that both include the `.h` file and contain all the hints have highest priority, those that contain all the hints have the next highest priority, and all other files have the lowest priority.

Once the `.c` files have been selected and ordered, they are processed as described in Section III-D, as though they all occurred in the same patch but without mutations. To reduce costs for header files that are used in a very large number of `.c` files, if more than 100 `.c` files are selected, we use only the configuration generated by `allyesconfig`, rather than considering any relevant configurations in `configs` directories. This approach incurs a small risk of false positives, *i.e.*, changed lines reported as not compiled when JMake could have compiled them with a more specialized configuration. Thus, the threshold is user-configurable. In our experiments, we encounter such false positives for 23 file instances, out of 21012 file instances considered. Success for a `.h` file is achieved when all of the `.h` file mutations have been found in at least one `.c` file that can be successfully compiled for at least one architecture. Note that the `.c` files found in the patch can also be included in the set of `.c` files selected for compiling the `.h` files and thus processed a second time. This situation may result in unnecessary overhead in some cases, but allowing the possible duplication simplifies the implementation.

[5]Other things could be taken as hints, such as the name of modified type definitions or functions. We have not yet explored these options.

IV. Janitors

A goal of our approach is to ease the task of kernel janitors by providing feedback on the reliability of the process of compile-testing their changes. In order to assess whether we are able to meet that goal in practice, we need to identify patches made by janitors and these patches' compilation properties. Currently, we are not aware of any formal definition of what a janitor is. Indeed, a single person can move in and out of a janitor role over time or combine janitor work with his work on a subsystem for which he is primarily responsible.

We consider that the main characteristic of a janitor is that such a developer works on the code base in a breadth-first way, touching many files and many subsystems, and doing about the same small amount of work on each one. The number of files and number of changes on each file are straightforward to collect. More complex is the notion of subsystem. Evidence such as the directory structure has not evolved in a uniform way, and does not take into account, *e.g.*, the possibility that relevant header files may be in the top-level `include` subdirectory and relevant architecture-specific support files may be in the `arch` subdirectory. We turn instead to the Linux kernel file `MAINTAINERS` that lists the maintainers associated with various collections of files, and consider that an entry in that file may represent a subsystem. `MAINTAINERS` entries, however, may be overlapping and the boundaries between them may correspond to individual maintainer interests rather than a precise definition of subsystem. As a somewhat more coarse-grained measure, we also consider the mailing lists that are designated to receive patches on various files; often multiple related entries in the `MAINTAINERS` file will refer to the same mailing lists.

Because of the approximativeness of our definition of subsystem, our preliminary proposal uses the subsystem information in a coarse-grained way. Namely, we first select a set of developers who over a period of time have contributed a minimum number of patches, to a minimum number of subsystems, and relevant to a minimum number of mailing lists. We also allow a limited amount of maintainer activity, but do not consider patches for which the developer is a maintainer in our further analysis. The thresholds that we use in our experiments, in which we try to identify janitors based on contributions between Linux v3.0, released in July 2011, and Linux v4.4, our reference version, are shown in Table I; the starting date is arbitrary, but was chosen to include activity over a number of years. Then, we rank the

TABLE I
THRESHOLDS ON JANITOR ACTIVITY BETWEEN LINUX V3.0 AND V4.4

# patches	≥ 10
# subsystems	≥ 20
# lists	≥ 3
# maintainer patches	$\leq 5\%$

TABLE II
JANITORS IDENTIFIED USING OUR CRITERIA

	patches	subsystems	lists	maintainer	file cv
Javier Martinez Canillas	118	61	30	0%	0.25
Luis de Bethencourt	104	56	31	0%	0.41
Dan Carpenter (T)	1554	400	146	0%	0.43
Julia Lawall (T)	653	255	93	0%	0.67
Shraddha Barke (I)	160	21	14	0%	0.72
Joe Perches (T)	1078	530	158	2%	0.81
Axel Lin	1044	142	49	0%	0.92
Daniel Borkmann	121	25	15	0%	1.29
Fabio Estevam	790	95	42	0%	1.29
Jarkko Nikula	173	30	14	0%	1.35

developers satisfying the thresholds according to the *coefficient of variation* (cv) of the number of patches contributed by the developer that touch each file that is ever touched by the developer. The cv is computed as the standard deviation of a set of numbers divided by their mean. This value gives information analogous to the standard deviation, but abstracts away from the number of patches involved, represented by the mean. The goal of this ranking is to highlight developers who have contributed roughly the same number of changes to each file, rather than concentrating on only a few files, on which they may become aware of configurability pitfalls.

For the purpose of our experiments, we need a sufficiently large set of janitor patches among the patches that we consider. Thus, we furthermore consider only developers who have contributed at least 20 patches between Linux v4.3 and Linux v4.4, the period that we study in our evaluation. We arbitrarily take the top 10 developers indicated by the ranking. These developers and their characterization in terms of the above metrics are shown in Table II. We note that three of the developers, indicated by "(T)", are developers of static analysis tools, who use the developed tools in their work on the Linux kernel. One of the developers, indicated by "(I)", was an applicant for an internship program designed to bring new developers into kernel development during part of the considered time period. In the three months between Linux v4.3 and v4.4, the 10 developers have contributed 591 patches.

V. EVALUATION

Our evaluation considers the patches accepted into the Linux kernel between version v4.3, released in November 2015, and v4.4, released in January 2016, covering a period of roughly 3 months. We first describe our experimental settings, then assess the benefits of JMake, then measure JMake's run time performance, and finally consider a minor limitation.

A. Experimental settings

We use `git log` with the options `-w --diff-filter=M --no-merges` to obtain the patches associated with all of the commits from Linux v4.3 to v4.4. These options ignore changes in whitespace (`-w`), only take commits that modify files (`--diff-filter=M`), and ignore commits that perform only merges. This gives 12,946 patches. In these patches, we only consider changes that affect `.c` and `.h` files. We furthermore ignore files in the `Documentation`, `scripts`, and `tools` directories, as these are typically not intended to run at the kernel level. 2099 of the 12,946 patches are completely ignored for these reasons.

TABLE III
CHARACTERISTICS OF ALL PATCHES AND OF JANITOR PATCHES

	All patches	Janitor patches
`.c` files only	7614 (70%)	514 (87%)
`.h` files only	631 (5%)	16 (2%)
both `.c` and `.h` files	2602 (23%)	60 (10%)

We process the remaining 11,057 patches iteratively using 25 parallel processes on a 48-core AMD Opteron 6172, having 2.1 GHz CPUs, 12 512KB L2 caches, and 251G RAM. 25 copies of the Linux kernel git tree[6] were stored in a 126GB in-memory file system, to avoid disk access bottlenecks. All compilation steps, including the generation of any temporary files, takes place in this in-memory file system. We chose to use only 25 processes to avoid running out of space in this filesystem during the experiments. Note that this amount of computing power is only needed for large-scale experiments. A developer would typically check at most a few patches at once, affecting a few files, which could be done easily on a standard laptop. The processing of each patch first cleans the git repository, using `git clean -dfx`, and then uses `git reset --hard` to check out the snapshot of the source code resulting from applying the patch that is to be processed. Then, the mutation engine and compile checker are used, as described in Section III. Compilations are limited to at most 50 `.c` files at a time to avoid generating files that exceed the size of the in-memory file system.

Our evaluation considers both the complete set of patches and the subset of patches contributed by the developers identified as janitors in Section IV. The former shows the conditions that can potentially affect these janitors, while the latter shows the conditions that have affected them, or would affect them when using JMake. The patches involved in the two cases are characterized in Table III. We observe that the percentage of patches affecting `.h` files is much lower for the janitors, suggesting a difference between the kinds of changes performed by janitors and the complete set of patches.

In this section, we refer to a *file instance* as an instance of a file at the time of a given commit.

B. Benefits of JMake

The two main features provided by JMake are the choice of target architecture for the compilation and the use of mutation to detect lines that are subjected to compilation. We now assess the benefits of these features.

[6]https://kernel.googlesource.com/pub/scm/linux/kernel/git/torvalds/linux

Choice of architecture: JMake provides automatic identification of architectures that are relevant to the files affected by a patch. Ideally, all files not in the `arch` directory would compile for the x86_64 architecture (x86, 64-bit), which typically found on the machines used by developers today, and thus there would be no question about what make command to use. For the complete set of patches, however, we found that 365 non-`arch` `.c` file instances do not compile for the x86_64 architecture, but do compile successfully for at least one other architecture, which in turn subjects at least part of the changes to compilation. Likewise, 75 non-`arch` `.h` file instances do not benefit from compilation for the x86_64 architecture, but do benefit from compilation for at least one other architecture. For the subset of janitor patches, 38 non-`arch` `.c` file instances do not compile for x86_64 but do benefit from compilation for some other architecture. On the other hand, compilation for x86_64 is beneficial for all of the `.h` files for which JMake is able to cause any of the changed lines to be subjected to compilation.

The frequency of usefulness of compilation for various architectures in our experiments is as follows. 96% of the complete set of file instances for which some lines are subjected to compilation benefit from compilation for the x86_64 architecture, which is the architecture of our host machine and thus the first architecture tried by JMake. This number is 95% for the janitor file instances. In both cases, when more architectures need to be considered, the next most frequently beneficial architecture is arm. 17 other architectures were found to be beneficial for some file instance in our experiments. Note, however, that this result depends on the order in which the architectures are considered, because some changes may be compileable equally well for multiple architectures. In our experiments, the janitor patches only benefited from compilation for four other architectures: powerpc, mips, blackfin, and parisc.

Finally, we consider the choice between `allyesconfig` and a more specific prepared configuration from a `configs` directory. We can compile all changed lines successfully for 9158 (84%) of patches when always using the configuration generated by `allyesconfig`, for various architectures, while we succeed for 9259 (85%) when also taking the `configs` configurations into account.

Properties of mutations: JMake uses mutations to check whether all changed lines are subjected to compilation. JMake tries to limit the number of mutations, to reduce the number of sites that the user must check manually when some lines are reported to be not subjected to compilation. For `.c` files, 82% of file instances require only one mutation and 95% require three or fewer. There may arise a large number of mutations when there are many changes in macro definitions. One `.c` file instance, drivers/clk/bcm/clk-bcm2835.c in commit 41691b8, requires over 200 mutations. For `.h` files, 75% of file instances require only one mutation, and 92% require three or fewer. Typically, Linux kernel `.h` files define more macros and contain more ifdefs than `.c` files, both of which may induce the need for mutations.

TABLE IV
REASONS WHY SOME CHANGED LINES ARE NOT SUBJECTED TO THE COMPILER

	affected file instances
change under `#ifdef` variable not set by allyesconfig	5
change under `#ifdef` variable never set in the kernel	5
change under `#ifdef` MODULE	3
change under `#ifndef` or `#else`	2
change under both `#ifdef` and `#else`	1
change under `#if` 0	1
change in unused macro	5

For janitor patches, for `.c` files, 91% of file instances require only one mutation and 98% require three or fewer. For `.h` files, 84% of file instances require only one mutation, and 93% require three or fewer. These numbers are greater than the overall numbers, implying smaller numbers of mutations required, again suggesting a difference in the kinds of changes performed by janitors. For janitor file instances, for both `.c` and `.h` files, at most 15 mutations are sufficient.

Benefits of mutations for `.c` files: JMake's mutations provide a benefit when they enable JMake to detect the insidious case when compilation succeeds, but the compiler has not actually considered all of the changed lines. In the overall patch set, for 14,097 (88%) `.c` file instances, JMake confirms that this situation does not arise; all changed lines are subjected to the compiler in the first compilation that produces no error messages. For 415 (3%) of the `.c` file instances, when only attempting `allyesconfig`, a first compilation produces no error message, but does not subject all of the changed lines to compilation. For 54 of these file instances, by considering multiple architectures, JMake is able to subject all of the changed lines to compilation, involving between 1 and 10 extra compilation steps. For the janitor patches, when only attempting `allyesconfig`, 21 `.c` file instances compile for some architecture without errors, but not all changed lines are subjected to compilation. For all of these file instances, none of the configurations tried by JMake leads to compilation of all of the changed lines.

For the above 21 janitor file instances, we have studied the code to determine why some lines are not subjected to compilation. The reasons for these failures are summarized in Table IV (one file instance exhibits multiple issues). Several cases involve an ifdef on `MODULE`, determining whether the code should be compiled as a kernel module. JMake could cause these lines to be compiled by additionally using `make allmodconfig`, at the cost of nearly doubling the set of configurations considered. For 6 of the file instances, the reason why some lines are not subjected to the compiler could be detected from the context lines in the patch code, while for the others a more thorough study of the affected file was required. These issues affect patches contributed by 5 of our 10 identified janitors.

Benefits of mutations for `.h` files: The situation for `.h` files is somewhat different, because they cannot be compiled

directly. In the ideal case, compiling the `.c` files modified by a patch causes all of the lines changed in the `.h` files modified by the patch to be subjected to compilation, and no further work is required. This is the case for 2888 (66%) of the `.h` file instances in the overall set of patches, and for 67 (76%) of the `.h` file instances among the janitor patches.

For the remaining patches that affect `.h` files, we need to find `.c` files that will cause their changed code to be compiled. The effect of this process depends greatly on the set of `.c` files chosen and the order in which they are considered. To abstract away from this issue, we consider the ideal case, where the only compilations attempted are the ones that actually subject at least one `.h` file changed line to compilation. In the overall set of patches, 1431 (33%) `.h` file instances need the selection and compilation of at least one additional `.c` file. For 730 (16%) `.h` files, JMake ultimately causes all of the changed lines to be subjected to compilation, using between 1 and 11 `.c` file compilations. For the 117 (2%) `.h` files where some changed lines are never subjected to compilation, JMake performed between 1 and 12 `.c` file compilations. Similar percentages are obtained for the janitor file instances, but at most 3 compilations per file instance are required.

Summary: Ultimately, JMake causes all changed lines to be subjected to compilation for 85% (9259) of the overall set of patches, and 88% (523) of the janitor patches.

C. Performance

The running time of JMake is dominated by the time to create a new configuration (`make allyesconfig`, etc.), the time to generate the `.i` files, and the time to generate a `.o` file when a mutant is detected in a `.i` file. Figure 4 shows the amount of time per invocation for each of the operations for the complete set of patches between Linux v4.3 and v4.4 as a cumulative distribution function (CDF); for each amount of time t, shown on the x-axis, the y-axis value indicates the percentage of invocations that complete in time less than or equal to t.

The time per invocation for creating a new configuration (Figure 4a) is 5 seconds or less for all of the invocations.

The time per invocation for generating the `.i` files (Figure 4b) is 15 seconds or less for 98% of the invocations, but other invocations are longer, up to 22 seconds. The variability is due to the number and contents of the files being processed. While typically a large part of the cost of a kernel `make` invocation is the set up time, some kernel files may be very large, which may also contribute to the variation in execution time.

The time per invocation for generating the `.o` files (Figure 4c) is 7 seconds or less for 97% of the invocations, with a maximum time of 15 seconds for almost all files. Unlike the `.i` files where we group the processing of up to 50 files, the `.o` files are generated individually, only when a mutant is detected in a `.i` file, leading to more uniform times. The largest times, of over 6000 seconds (1.6 hours), are for the file `arch/powerpc/kernel/prom_init.c`, which is modified by three of our considered patches. Compilation of this single file triggers compilation of the entire kernel, whether or not JMake is used.

The impact of these running times is determined by the number of invocations performed per patch. The latter are determined by the number of architectures that are considered, and by the number of `.c` files considered. 87% of the patches require only one choice of configuration, but others require up to 92; note that we may consider each configuration twice, first for the `.c` files and then for the `.h` files. Similarly, 85% of the patches require only one make invocation to generate `.i` files, but the largest number required is 299, which can occur when a patch touches a `.h` file that is referenced by many `.c` files. The numbers for `.o` files are similar, with 72% of patches requiring to generate only one `.o` file, and 88% requiring 5 or fewer. The worst case involves generating 202 `.o` files. Figure 5 shows the resulting overall running time of JMake. 82% of the patches require 30 seconds or less, while 95% require at most one minute.

Considering the patches contributed by our 10 identified janitors only, Figure 6 shows the overall running time of JMake. The curve has the same shape as in Figure 5 for the complete set of patches, but does not contain the highest values, suggesting that the patches contributed by janitors require fewer iterations of JMake to cause all of the changed lines to be subjected to the compiler. Indeed, the longest running time for the janitor patches is 1080 seconds (around 18 minutes), while it is over 6000 seconds for the complete set of patches. As over 90% of patches again require less than a minute, we believe that JMake has acceptable performance for janitor use.

D. Limitations

Our approach relies on first mutating the files changed by a patch and then creating `.i` files to determine whether the changed lines will be subjected to compilation. Unfortunately, the kernel Makefile itself compiles some code from the kernel source tree in order to carry out any compilation task, including generation of `.i` files. It is thus impossible to mutate the files involved in this preliminary compilation step. Among the complete set of patches between Linux v4.3 and v4.4, 317 patches, amounting to 2% of the total number of patches, affect a total of 411 file instances that are suspected of being involved in this compilation setup process and thus cannot be treated by JMake. For the janitor patches, this issue affects only one file in one patch. It could be interesting to see if some of the preliminary compilation performed by the kernel Makefile is actually not necessary for producing `.i` files, which could at least enable providing partial information in these cases, even if it is not possible to produce the corresponding `.o` files to fully validate the compilation process.

VI. Related Work

A number of recent works have focused on the problems arising from the use of conditional compilation in the Linux kernel. The Undertaker [6] analyzes the interdependencies between configuration variables and identifies inconsistencies

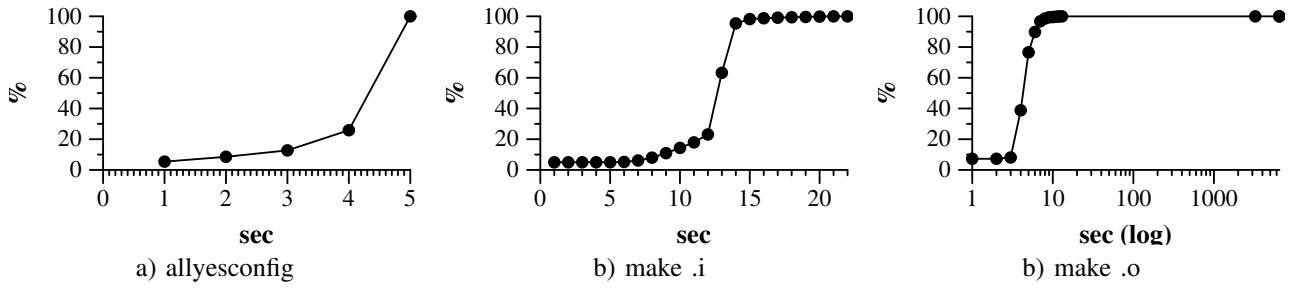

Fig. 4. CDFs of the running times of the main steps of JMake

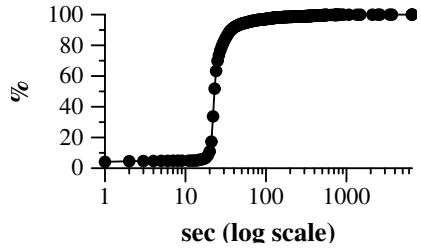

Fig. 5. CDF of the overall running time of JMake

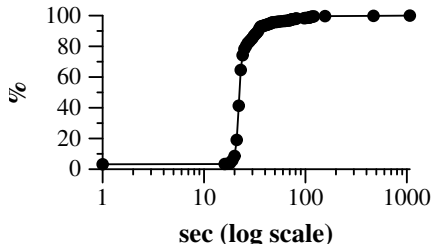

Fig. 6. CDF of the overall running time of JMake on janitor patches

in their usage, such as blocks of code that are *undead* or *dead*, *i.e.*, that depend on a composition of values of configuration variables that represents a tautology or a contradiction, respectively. Nadi *et al.* [7] explore what kinds of code changes cause these faults to be introduced, and how they are fixed. Abal *et al.* [8] study bugs in Linux kernel code that manifest only in a subset of the possible configurations. We focus instead on understanding the degree to which conditional compilation affects the compilation of changes made in practice, when using standard configurations. We also propose a tool that gives feedback to the developer about whether a given configuration causes the compiler to overlook some of his changed code.

JMake relies on the ability to obtain configuration files that provide high statement coverage. The Linux kernel provides the `allyesconfig` make target for generating such a configuration. The resulting configuration, however, is forced to make some choices and thus does not include all lines of code. Vampyr [5] and Troll [4], provide strategies to construct a set of configurations that will maximize the coverage of a static analysis tool, such as a compiler, on a changed file. Sampling strategies have also been used for this purpose; Medeiros *et al.* [9] review 10 such strategies and compare their efficiency in fault finding. These approaches could be used in extending the JMake approach to software for which configurations with high statement coverage are not easily available.

Other more general work on issues related to software configurability includes studies of how the C preprocessor is used in open source and industrial software [10], studies of how the Linux configuration model evolves [11], [12] and the development of tools for processing code that contains ifdefs [13], [14].

On the practical side, the Linux kernel 0-day build testing service [3] devotes a large amount of computing resources to regularly run a large number of checks on all commits to the Linux kernel, including compiling the kernel with random configurations. These checks make it likely that any problem that the developer overlooks due to insufficient compile testing will be detected after the change is submitted. Still, relying on this service introduces a delay into the development process and receiving an error message from such a service can be discouraging. JMake, on the other hand, requires only standard computing resources, allowing it to be run by the developer himself, and gives immediate feedback on any changes that are not checked by the compiler in the chosen configuration.

Mutation testing has a long history in the context of fault injection tools [15]. Mutations typically target the source code, while other fault injection techniques target the compiler [16]. The developers of the Devil device driver interface definition language used mutations in the device driver interface to show that their language checker was able to detect erroneous specifications before they were deployed into an operating system kernel [17]. In our case, we use mutations not to generate and detect errors in the source code, but to unambiguously track source code as it passes through the C preprocessor into the set of code lines that will be checked by the compiler.

VII. Conclusion and Future Work

A major stumbling block in contributing to the Linux kernel is to know whether a change is correct, or indeed if it even compiles. In this paper, we have presented a tool to help answer the latter question. Using heuristics, JMake takes care of the drudgery of identifying the architecture(s) for which the code can be compiled, and then uses mutations to check

whether a set of successful compilations actually subjects all of the changed lines to the compiler. We have shown that JMake is able to certify that 85% of the full set of patches between Linux v4.3 and v4.4 satisfy the property that every changed line has been subjected to at least one invocation of the compiler, and that this is the case for 88% of the patches from our identified set of janitors. For the remaining patches, JMake provides feedback on which lines have not been subjected to the compiler for the tested configurations, and thus require further attention. The conclusion for the janitor is both encouraging and cautioning. Indeed, most of the time, *i.e.* for 79% of the overall set of patches, a single successful compilation is sufficient to ensure that the changed lines are subjected to the compiler. Still, our results show that this is not always the case, and JMake can help protect against any errors that may creep into the affected code, even from the most careful janitor.

While JMake runs in 30 seconds or less in our experiments on many patches, on others it requires hundreds of seconds or more, which may not be acceptable to kernel janitors. JMake typically has its longest running times when processing `.h` files that are included in many `.c` files. JMake uses some heuristics based on the names of changed macros to prioritize `.c` files that are more likely to trigger compilation of the changed lines, but it could be beneficial to investigate other heuristics to take into account.

Currently, JMake never succeeds for a file containing a change that comprises changes under both an ifdef and the corresponding else, unless the tested configuration variable is architecture dependent, and often fails on code under ifndef, because `make allyesconfig` tries to set configuration variables to "yes" rather than "no". JMake could be complemented with more sophisticated configuration generation techniques, as presented in Section VI, to obtain better results in such cases. JMake would still provide the added value of feedback about changed lines. Alternatively, JMake could simply detect the issue and ask for user assistance, which could save running time by avoiding the exploration of unpromising cases.

JMake has been primarily designed with the needs of Linux kernel developers in mind. nevertheless, there are many other highly configurable systems. The main Linux-specific aspects of our approach are the availability of the `allyesconfig` make target and the focus on the `arch` subdirectory for identifying alternate configurations relevant to particular file. As noted in Section VI, software-project independent approaches have been developed for generating interesting configurations. Integrating these approaches into JMake could enable extending our approach to other software systems.

Finally, our current characterization of janitors involves thresholds that have been chosen arbitrarily. In the future, we would like to explore other means of characterizing janitors, to get a better understanding of the specific properties of the work the perform and the challenges they face. This understanding may then motivate new tools to better address their needs. Better tool support can free up more janitor time for fixing bugs and faults in the code, and can encourage more developers to address janitorial issues.

Availability

JMake is available under the licence GPLv2 at `http://jmake-release.gforge.inria.fr`.

Acknowledgments

This work was supported in part by OSADL as part of the SIL2LinuxMP project.

REFERENCES

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP*, 2001, pp. 73–88.

[2] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall, "Faults in Linux 2.6," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 4:1–4:40, 2014.

[3] M. Kerrisk, "Ks2012: Kernel build/boot testing," 2012, https://lwn.net/Articles/514278/.

[4] V. Rothberg, C. Dietrich, A. Ziegler, and D. Lohmann, "Towards scalable configuration testing in variable software," in *GPCE*, 2016, pp. 156–167.

[5] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90,000 #ifdefs issue," in *USENIX Annual Technical Conference*, 2014, pp. 421–432.

[6] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann, "Revealing and repairing configuration inconsistencies in large-scale system software," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 5, pp. 531–551, 2012.

[7] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann, "Linux variability anomalies: what causes them and how do they get fixed?" in *MSR*, 2013, pp. 111–120.

[8] I. Abal and A. W. Claus Brabrand and, "42 variability bugs in the Linux kernel: a qualitative analysis," in *ASE*, 2014, pp. 421–432.

[9] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *ICSE*, Austin, Texas, 2016, pp. 643–654.

[10] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel, "Preprocessor-based variability in open-source and industrial software systems: An empirical study," *Empirical Software Engineering*, vol. 21, no. 2, pp. 449–482, 2016.

[11] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux kernel variability model," in *International Software Product Line Conference*, 2010, pp. 136–150.

[12] L. Teixeira Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, and P. Borba, "Coevolution of variability models and related artifacts: a case study from the Linux kernel," in *International Software Product Line Conference*, 2013, pp. 91–100.

[13] P. Gazzillo and R. Grimm, "SuperC: Parsing all of C by taming the preprocessor," in *PLDI*, Beijing, China, 2012, pp. 323–334.

[14] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *OOPSLA*, 2011, pp. 805–824.

[15] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 44:1–44:55, Feb. 2016.

[16] G. Li, Q. Lu, and K. Pattabiraman, "Fine-grained characterization of faults causing long latency crashes in programs," in *DSN*, 2015, pp. 450–461.

[17] L. Réveillère and G. Muller, "Improving driver robustness: An evaluation of the Devil approach," in *DSN*, 2001.