Impact of Tool Support in Patch Construction

Anil Koyuncu
SnT, University of Luxembourg - Luxembourg

Tegawendé F. Bissyandé
SnT, University of Luxembourg - Luxembourg

Dongsun Kim
SnT, University of Luxembourg - Luxembourg

Jacques Klein
SnT, University of Luxembourg - Luxembourg

Martin Monperrus
Inria, University of Lille - France

Yves Le Traon
SnT, University of Luxembourg - Luxembourg

ABSTRACT

In this work, we investigate the practice of patch construction in the Linux kernel development, focusing on the differences between three patching processes: (1) patches crafted entirely manually to fix bugs, (2) those that are derived from warnings of bug detection tools, and (3) those that are automatically generated based on fix patterns. With this study, we provide to the research community concrete insights on the practice of patching as well as how the development community is currently embracing research and commercial patching tools to improve productivity in repair. The result of our study shows that tool-supported patches are increasingly adopted by the developer community while manually-written patches are accepted more quickly. Patch application tools enable developers to remain committed to contributing patches to the code base. Our findings also include that, in actual development processes, patches generally implement several change operations spread over the code, even for patches fixing warnings by bug detection tools. Finally, this study has shown that there is an opportunity to directly leverage the output of bug detection tools to readily generate patches that are appropriate for fixing the problem, and that are consistent with manually-written patches.

CCS CONCEPTS

•**Software and its engineering →Software maintenance tools;** *Software configuration management and version control systems; Software version control;*

KEYWORDS

Repair, Debugging, Patch, Linux, Empirical, Tools, Automation

ACM Reference format:
Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2017. Impact of Tool Support in Patch Construction. In *Proceedings of International Symposium on Software Testing and Analysis, Santa Barbara, California USA, July 2017 (ISSTA'17),* 12 pages.
DOI: 10.1145/3092703.3092713

ISSTA'17, Santa Barbara, California USA

DOI: 10.1145/3092703.3092713

1 INTRODUCTION

Patch construction is a key task in software development. In particular, it is central to the repair process when developers must engineer change operations for fixing the buggy code. In recent years, a number of tools have been integrated into software development ecosystems, contributing to reducing the burden of patch construction. The process of a patch construction indeed includes various steps that can more or less be automated: bug detection tools for example can help human developers characterize and often localize the piece of code to fix, while patch application tools can systematize the formation of concrete patches that can be applied within an identified context of the code.

Tool support however can impact patch construction in a way that may influence acceptance or that focuses the patches to specific bug kinds. The growing field of automated repair[20, 24, 29, 33], for example, is currently challenged by the nature of the patches that are produced and their eventual acceptance by development teams. Indeed, constructed patches must be applied to a code base and later maintained by human developers.

This situation raises the question of the acceptance of patches within a development team, with regards to the process that was relied upon to construct them. The goal of our study is therefore to identify different types of patches written by different construction processes by exploring patches in a real-world project, to reflect on how program repair is conducted in current development settings. In particular, we investigate how advances in static bug detection and patch application have already been exploited to reduce human efforts in repair.

We formulate research questions for comparing different types of patches, produced with varying degrees of automation, to offer to the community some insights on i) whether tool-supported patches can be readily adopted, ii) whether tool-supported patches target specific kinds of bugs, and iii) where further opportunities lie for improving automated repair techniques in production environments.

In this work, we consider the Linux operating system development since it has established an important code base in the history of software engineering. Linux is furthermore a reliable artifact [17] for research as patches are validated by a strongly hierarchical community before they can reach the mainline code base. Developers involved in Linux development, especially maintainers who are in charge of acknowledging patches, have relatively extensive experience in programming. Linux's development history constitutes a valuable information for repair studies as a number of tools have been introduced in this community to automate and systematize

various tasks such as code style checking, bug detections, and systematic patching. Our analysis unfolds as an empirical comparative study of three patch construction processes:

- **Process H:** In the first process, developers must rely on a bug report written by a user to understand the problem, locate the faulty part of source code, and manually craft a fix. We refer to it as *Process H*, since all steps in the process appear to involve **H**uman intervention.
- **Process DLH:** In the second process, static analysis tools first scan the source code and report on lines which are likely faulty. Fixing the reported lines of code can be straightforward since the tools may be very descriptive on the nature of the problem. Nevertheless, dealing with static debugging tools can be tedious for developers with little experience as these tools often yield too many false positives. We refer to this process as *Process DLH*, since **D**etection and **L**ocalization are automated but **H**uman intervention is required to form the patch.
- **Process HMG:** Finally, in the third process, developers may rely on a systematic patching tool to search for and fix a specific bug pattern. We refer to this process as *Process HMG*, since **H**uman input is needed to express the bug/fix patterns which are **M**atched by a tool to a code base to **G**enerate a concrete patch.

We ensure that the collected dataset does not include patch instances that can be attributed to more than one of the processes described above. Our analyses have eventually yielded a few implications for future research:

Acceptance of patches: development communities, such as the Linux kernel team, are becoming aware of the potential of tool support in patch construction i) to gain time by prioritizing engineering tasks and ii) to attract contributions from novice developers seeking to join a project.

Kinds of bugs: Tool-supported patches do not target the same kinds of bugs as manual patches. However, we note that patches fixing warnings outputted by bug detection tools are already complex, requiring several change operations over several lines, hunks and even files of code.

Opportunities for automated repair: We have performed preliminary analyses which show that bug detection tools can be leveraged as a stepping stone for automated repair in conjunction with patch generation tools, to produce patches that are consistent with human patches (for maintenance), correct (derived from past experience of fixing a specific bug type) and thus likely to be rapidly accepted by development teams.

2 BACKGROUND

Linux is an open-source operating system that is widely used in environments ranging from embedded systems to servers. The heart of the Linux operating system is the Linux kernel, which comprises all the code that runs with kernel privileges, including device drivers and file systems. It was first introduced in 1994, and has grown to 14.3 million lines of C code with the release of Linux 4.8 in Oct. 2016.[1] All data used in this paper are related to changes propagated to the mainline code base until Oct. 2, 2016[2].

A recent study has shown that, for a collection of typical types of faults in C code, the number of faults is staying stable, even though the size of the kernel is increasing, implying that the overall quality of the code is improving [37]. Nevertheless, ensuring the correctness and maintainability of the code remains an important issue for Linux developers, as reflected by discussions on the kernel mailing list [42].

The Linux kernel is developed according to a hierarchical open source model referred to as Benevolent dictator for life (BDFL) [50], in which anyone can contribute, but ultimately all contributions are integrated by a single person, Linus Torvalds. A Linux kernel maintainer receives patches related to a particular file or subsystem from developers or more specialized maintainers. After evaluating and locally committing them, he/she propagates them upwards in the maintainer hierarchy eventually towards Linus Torvalds.

Finally, Linux developers are urged to "solve a single problem per patch"[3], and maintainers are known to enforce this rule as revealed by discussions on contributors' patches in the Linux Kernel Mailing List (LKML) [42] archive.

Recently, the development and maintenance of the Linux kernel have become a massive effort, involving a huge number of people. 1,731 distinct commit authors have contributed to the development of Linux 4.8[4]. The patches written by these commit authors are then validated by the 1,142 *maintainers* of Linux 4.8[5], who are responsible for the various subsystems.

Since the release of Linux 2.6.12 in June 2005, the Linux kernel has used the source code management system `git` [13]. The current Linux kernel git tree [47] only goes back to Linux 2.6.12, and thus we use this version as the starting point of our study. Between Linux 2.6.12 and Linux 4.8 there were 616,291 commits, by 20,591 different developers[6]. These commits are retrievable from the git repository as *patches*. Basically, a patch is an extract of code, in which lines beginning with - are to be removed lines beginning with + are to be added.

The Linux kernel community actively uses the Bugzilla [19] issue tracking system to report and manage bugs. As of November 2016, over 28 thousands bug reports were filed in the kernel tracking system, with about 6,000 marked as highly severe or even blocking.

The Linux community has also built, or integrated, a number of tools for improving the quality of its source code in a systematic way. For example, The mainline code base includes the coding style checker *checkpatch*, which was released in July 2007, in Linux 2.6.22. The use of checkpatch is supported by the Linux kernel guidelines for submitting patches[7], and checkpatch has been regularly maintained and extended since its inception. Sparse [49] is another example of the tools built by Linus Torvalds and colleagues to enforce typechecking.

Commercial tools, such as Coverity [44], also often help to fix Linux code. More recently, researchers at Inria/LiP6 have developed the Coccinelle project [25] for Linux code matching and transformation. Initially, the project was designed to help developers perform collateral evolutions [35]. It is now intensively used by Linux developers to apply fix patterns to the whole code base.

[1] Computed with David A. Wheeler's 'SLOCCount'.
[2] Kernel's Git HEAD commit id is `c8d2bc9bc39ebea8437fd974fdbc21847bb897a3`.
[3] see `Documentation/SubmittingPatches` in linux tree.
[4] Obtained using `git log v4.7..v4.8 | grep ^Author | sort -u | wc -l`, without controlling for variations in names or email addresses.
[5] Obtained using `grep ^M: MAINTAINERS | sort -u | wc -l` without controlling for variations in names or email addresses.
[6] Again, we have not controlled for variations in names or email addresses.
[7] `Documentation/SubmittingPatches` in the Linux tree.

3 METHODOLOGY

Our objective is to empirically check the impact of tool support in the patch construction process in Linux. To achieve this goal, we must collect a large, consistent and clean set of patches constructed in different processes. Specifically, we require:

(1) patches that have been a-priori manually prepared by developers based on the knowledge of a potential bug, somewhere in the code. For this type of patches, we assume that a user may have reported an issue while running the code. In the Linux ecosystem, such reporters are often kernel developers.
(2) patches that have been constructed by using the output of bug finding tools, which are integrated into the development chain. We consider this type of patches to be tool-supported, as debugging tools often provide reliable information on what the bug is (hence, how to fix it) and where it is located.
(3) patches that have been constructed, by a tool, based fully on change rules. Such fixes, validated by maintainers, are actually based on templates of fix patterns which are used to i) match (i.e., locate) incorrect code in the project and ii) generate a corresponding concrete fix.

3.1 Dataset Collection

To collect patches constructed via Process H, hereafter referred to as ***H patches***, we consider patches whose commits are explicitly linked to a bug report from the kernel bugzilla tracking system and any other Linux distributions bug tracking systems. We consider that such patches have been engineered manually after a careful consideration of the report filed by a user, and often after a replication step where developers dynamically test the software.

Until Linux 4.8, we have found 5,758 patches fixing defects described in bug reports. Unfortunately, for some of the patches, the link to its bug report provided in the commit log was not accessible (e.g., because of restriction in access rights of some Redhat bug reports or because the web page was no longer live). Consequently, we were able to collect 4,417 bug patches corresponding to a bug report (i.e., ~ 77% of H patches). Table 1 provides statistics on the bugs associated with those patches.

Table 1: Statistics on H patches in Linux Kernel.

Sevirity	# reports	# patches
Severe	965	1,052
Medium	2,961	3,163
Minor	138	136
Enhancement	47	66
Total	4,111	4,417

First, we note that the severity of most bugs (2,961, i.e., 72.0%) is medium, and H patches have fixed substantially more severe bugs (965, i.e., 23.5%) than minor bugs (138, i.e., 3.3%). Only 47 (1.1%) bug reports represent mere enhancements. Second, exploring the data shows that there is not always a 1 to 1 relationship between bug reports and patches: a bug report may be addressed by several patches, while a single patch may relate to several bug reports. Nevertheless, we note that 4,270 out of 5,265 (i.e., 89%) patches address a single bug report. Third, a large number of unique developers (1,088 out of 18,733= 6.95%) have provided H patches to fix user bug reports. Finally, H patches have touched about 17% (= 9,650/57,195) of files in the code base. Overall, these statistics suggest that the dataset of H patches is diverse as they are indeed written by a variety of developers to fix a variably severe set of bugs spread across different files of the program.

We identify patches constructed via Process DLH, hereafter referred to as ***DLH patches***, by matching in commit logs messages on the form "`found by <tool>`"[8] where `<tool>` refers to a tool used by kernel developers to find bugs. In this work, we consider the following notable tools, for static analysis:

- *checkpatch*: a coding style checker for ensuring some basic level of patch quality.
- *sparse*: an in-house tool for static code analysis that helps kernel developers to detect coding errors based on developer annotations.
- *Linux driver verification (LDV) project* : a set of programs, such as the Berkeley Lazy Abstraction Software verification Tool (BLAST) that solves the reachability problem, dedicated to improving the quality of kernel driver modules.
- *Smatch*: a static analysis tool.
- *Coverity*: a commercial static analysis tool.
- *Cppcheck*: an extensible static analysis tool that feeds on checking rules to detect bugs.

and for dynamic analysis:

- *Strace*: a tracer for system calls and signals, to monitor interactions between processes and the Linux kernel.
- *Syzkaller*: a supervised, coverage-guided Linux syscall fuzzer for testing untrusted user input.
- *Kasan*: the Linux Kernel Address SANitizer is a dynamic memory error detector for finding use-after-free and out-of-bounds bugs.

After collecting patches referring to those tools, we further check that commit logs include terms "bug" or "fix", to focus on bug fix patches. Table 2 provides details on the distribution of patches produced based on the output of those tools.

Table 2: Statistics on DLH patches in Linux Kernel.

Tool	# patches	Tool	# patches
checkpatch	292	sparse	68
LDV	220	smatch	39
coverity	84	cppcheck	14
strace	4	syzkaller	7
kasan	1		

Checkpatch and the *Linux driver verification project* tools are the most mentioned in commit logs. The *Coverity* commercial tool and the *sparse* internal tool also helped to find and fix dozens of bugs in the kernel. Finally, we note that static tools are more frequently referred to than dynamic tools.

HMG patches in Linux are mainly carried out by Coccinelle, which was originally designed to document and automate collateral evolutions in the kernel source code [35]. Coccinelle is built on an approach where the user guides the inference process using patterns of code that reflect the user's understanding of the conventions and design of the target software system [23].

Static analysis by Coccinelle is specified by developers who use control-flow sensitive concrete syntax matching rules [9]. Coccinelle provides a language, SmPL[9], for specifying search and transformations referred to as *semantic patches*. It also includes a transformation engine for performing the specified semantic patches. To

[8] We also use "`generated by <tool>`" since the commit authors also often refer to warnings as "generated by" a given tool.
[9] Semantic Patch Language.

avoid confusion with semantic patches in the context of automated repair literature, we will refer to Coccinelle-generated patches as *SmPL patches*.

```
1 @@
2 expression E;
3 constant c;
4 type T;
5 @@
6 -kzalloc(c * sizeof(T), E)
7 +kcalloc(c, sizeof(T), E)
```

(a) Example of SmPL templates.

```
1 void main(int i)
2 {
3
4 kzalloc(2 * sizeof(int), GFP_KERNEL);
5 kzalloc(sizeof(int) * 2, GFP_KERNEL);
6
7 }
```

(b) C code matching the template on the left. (iso-kzalloc.c).

Figure 1: Illustration of SmPL matching and patching.

Figure 1 illustrates a SmPL patch example. This SmPL patch is aimed at changing all function calls of *kzalloc* to *kcalloc* with a reorganization of call arguments. For more details on how SmPL patches are specified, we refer the reader to the project documentation[10]. Figure 2 represents the concrete Unix diff generated by Coccinelle engine and which is included in the patch to forward to mainline maintainers.

```
diff =
--- iso-kzalloc.c
+++ /tmp/cocci-output-52882-062587-iso-kzalloc.c
@@ -1,7 +1,7 @@
 void main(int i)
 {
- kzalloc(2 * sizeof(int), GFP_KERNEL);
- kzalloc(sizeof(int) * 2, GFP_KERNEL);
+ kcalloc(2, sizeof(int), GFP_KERNEL);
+ kcalloc(2, sizeof(int), GFP_KERNEL);
 }
```

Figure 2: Patch derived from the SmPL template in Figure 1a.

In some cases, the fix is not directly implemented in the SmPL patch (which is then referred to as SmPL *match*). Nevertheless, since each bug pattern must be clearly defined with SmPL, the associated fix is straightforward to engineer. Overall, we have collected 4,050 HMG patches mentioning "coccinelle" or "semantic patch" and applied to C code[11].

3.2 Research Questions

We now enumerate and motivate our research questions in the context of the three processes of patch construction:

RQ1 *How does the developer community react to the introduction of bug detection and patch application tools?*
With this research question, we check that the temporal distributions of patches in each patch construction process are in line with the upstream discussions for accepting patches. Such discussions may shed light on the proportions of tool-supported patches that are pushed by developers but that never get into the code base.

RQ2 *Who is using bug detection and patch application tools?*
In this research question, we investigate the profile of patch authors in the different patch construction processes.

RQ3 *What is the impact of patch construction process in the stability of patches?*
We investigate the stability, i.e., whether or not the patch is reverted after being propagated in the mainline tree, of accepted patches to highlight the reliability of each patch application tool within the community.

RQ4 *Do the patch construction processes target the same kind of bugs?*
We approximate the categorization of bugs with two metrics related to (1) the locality of the fixes as well as (2) the nature and number of change operators of the patch.

[10] http://coccinelle.lip6.fr/documentation.php

[11] We have controlled with a random subset of 100 commits that this grep-based approaches yielded indeed only relevant patches constructed by Coccinelle.

4 EMPIRICAL STUDY FINDINGS

4.1 Descriptive Statistics on the Data

We first provide statistics on how the different patch construction processes are used by developers over time and across project modules. Temporal distribution of patches may shed some light on the adoption of a patch construction process by kernel maintainers. Spatial distributions on the other hand may highlight the acceptance of a process based on the type (i.e., to some extent the critical nature) of the code to fix.

Temporal distribution of patches. We compute the temporal distribution of patches since Linux 2.6.12 (June 2005) until Linux 4.8 (October 2016) and outline them in Figure 3. Note that although Linux 2.6.12 was released in June 2005, a few commit patches in the code base pre-date this release date.

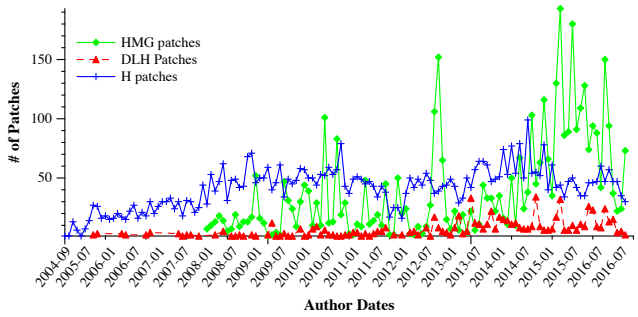


Figure 3: Temporal distributions of patches.

Overall, H patches are consistently applied over time with approximately 50 fixes per month. DLH patches have been very slow to take up. Indeed, the number of patches built based on bug finding tools has been narrow for several years, with a slight increase in recent years, partly due to the improvements made for reducing false positives. Finally, HMG patches have rapidly increased and now account for a significant portion of patches propagated to the mainline code base.

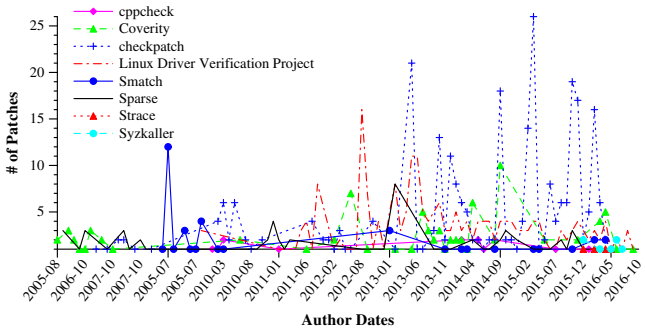
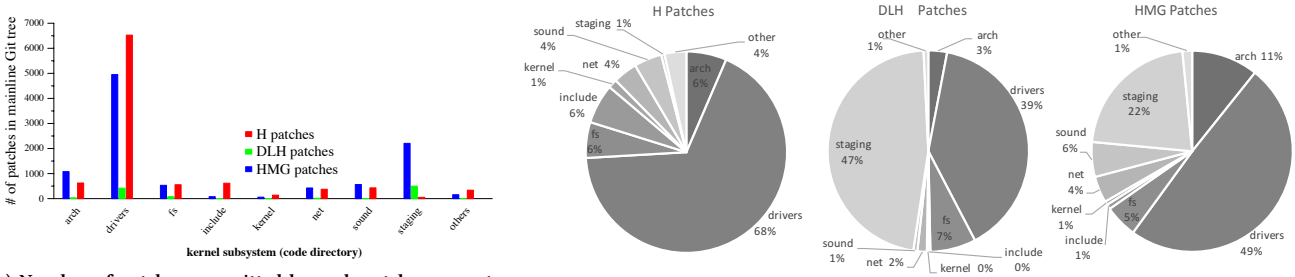


Figure 4: Temporal distributions of DLH patches broken down by tool.

Figure 4 represents the detailed temporal evolutions of DLH patches. Checkpatch, after a slow adoption, is now commonly used,



(a) Number of patches committed by each patch process to Linux's mainline code base.

(b) Percentage of patches per subsystem.

Figure 5: Spatial distribution of patches.

followed by Coverity, which regularly contributes to fix vulnerabilities and common operating system errors. Linux driver verification project tools and Smatch find fewer issues in mainline code base; such tools are indeed extensively used by developers before code is committed in the code base.

Spatial distribution of patches. We compute the spatial distribution of patches across Linux sub-systems. Linux Kernel's code is split into several folders, each roughly containing all code related to a specific sub-system such as file systems, device drivers, architectures, networking, etc. We investigate the scenarios of patches with regards to the folders where the files are changed and the results are shown in Figure 5. Most patches are targeted to device drivers code, and code in early development (i.e., in *staging/*[12]) that is not yet part of the running kernel. It is noteworthy that header code (*include/*), core kernel code (*kernel/*), and to some extent file system code (*fs/*), which have been extensively tested over the years, remain repaired mainly in an all-human process.

Driver code in general, and *drivers/staging/* code, in particular, appear to be the place where tool support is most prevalent. Percentages distribution in Figure 5b shows that half (46%) of DLH patches are targeted at *staging* code. 39% of DLH patches are applied to driver code. Several studies [12, 37, 38] have already shown that driver and staging code contained most kernel errors identified by static analysis tools. Similarly, HMG patches are applied in a large majority in drivers code and staging code.

4.2 Acceptance of Patches (RQ1)

We investigate the reaction of the developer community to the introduction of bug finding and patch application tools. To that end, we explore, first, the delays in integrating commits, then, the gaps between the number of patches proposed to the Linux community and those that are finally integrated.

Delay in commit acceptance. Kernel patches are change suggestions proposed by developers to maintainers who often need time to review them before propagating the changes to the mainline code base. Thus, depending on several factors — including the criticality of the bug, complexity of the fix, reliability of the suggested fix, and patch quality — there can be a more or less significant delay in commits.

We compute a delay in commit acceptance as the time difference between the author contribution date and the commit date (i.e., when the maintainer propagated the patch to mainline tree). Figure 6 shows the distribution of delays in the three different patch construction processes. Overall, H patches appear to be more[13] rapidly propagated (median = 2 days) than DLH (median = 4 days) and HMG patches (median = 4 days).

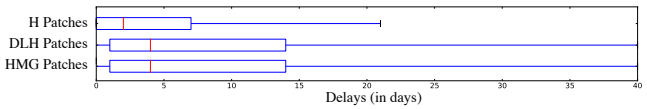
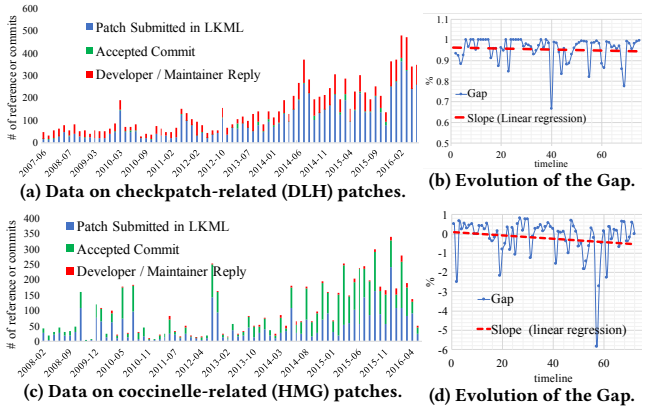


Figure 6: Delay in commit acceptance.

Gaps between discussion and acceptance trends. A patch represents the conclusion of an email exchange between the patch author and the relevant maintainers about the correctness of the proposed change. As the discussion takes place in natural language, it is difficult to categorize how the use of bug finding and patching tools are valued in the process. Nevertheless, we can use the mailing list to study the frequency at which developers specifically mention bug finding tools when a patch is first submitted. Then, we can correlate this frequency on a monthly basis with the corresponding statistics on accepted DLH patches related to the specific tools.



(a) Data on checkpatch-related (DLH) patches.

(b) Evolution of the Gap.

(c) Data on coccinelle-related (HMG) patches.

(d) Evolution of the Gap.

Figure 7: # of Patches submitted / discussed / accepted.

We have crawled all emails archived in the Linux Kernel Mailing List (LKML) using Scrapy[14]. We use heuristics to differentiate message replies from original mail content: we consider lines starting with '>' as part of a previous conversation. Finally, we naively search for the tool name reference in the message text. In total, we crawled[15] 1,601,606 original email messages and 885,814 reply

[12] *staging* is a sub-directory of *drivers* and contains code that does not yet meet kernel coding standards. We thus separate its statistics from statistics of *drivers*.

[13] We have checked with the Mann-Whitney Wilcoxon test that the difference between delay values is statistically significant.

[14] https://scrapy.org/, a framework for deploying and running spiders

[15] 7,510 entries were empty messages and were thus dropped out.

messages. As examples, we provide in Figures 7a and 7c the distributions per month of the number of patches that were submitted through LKLM mentioning *checkpatch* or coccinelle respectively, as well as the number of maintainer replies referencing those tools, and the number of related commits accepted into the mainline git tree. To ease observation, we compute in Figures 7b and 7d the integration gap as a percentage between the number of patches submitted to LKML and the number of patches that are eventually integrated. We draw the slope of the evolution of this gap over time. While checkpatch presents roughly the same gap, the gap is clearly reducing for coccinelle. We have computed the slope for the different sets of tool-supported patches and checked that it was negative for 3 out of 4 of the tools[16]: the gap is thus closing over time for most tool-supported processes.

Tool-supported patches (DLH and HMG alike) have been overall accepted at an increasing rate by Linux developers. Integration of such patches by maintainers remains, however, slower than that of traditional H patches.

4.3 Profile of Patch Authors (RQ2)

We investigate the speciality and commitment of developers who rely on patch application and bug finding tools to construct patches.

Speciality is defined as a metric for characterizing the extent to which a developer is focused on a specific subsystem. We compute it as the percentage of patches, among all her/his patches, which a developer contributes to a specific subsystem. Thus, *speciality* is measured with respect to each Linux code directory. We then draw, in Figure 8, the distributions of speciality metric values of developers for the different types of patches: e.g., for an automated patch applied to a file in a subsystem, we consider the commit author speciality w.r.t that subsystem.

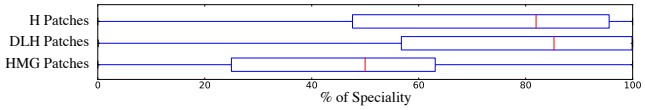


Figure 8: Speciality of developers Vs. Patch types.

H patches are mostly provided by specialized developers. This may imply that the developers focus on implementing specific functionalities over time. Similarly, DLH patches appear to be mostly applied by specialized developers (even slightly more specialized than those who made H patches). This finding is inline with the requirements for developers to be aware of the idiosyncrasies of the programming of a particular subsystem to validate the warnings of bug detection tools and sift through various false positives to produce patches that are eventually accepted by maintainers. HMG patches, on the other hand, are performed by developers on subsystem code which they are not known to be specialized on.

To measure developer *commitment*, we follow the approach of Palix et al. [38] and compute, for each developer, the product of (1) the number of patches (H, DLH or HMG) that have been integrated into Linux and (2) the number of days between the first patch and the last patch. This metric favours both developers who have contributed many patches over a short period of time and developers who have contributed fewer patches over a longer period of time: e.g., a developer who gets 10 commits integrated during one year, will have the same degree of commitment as another developer who gets 40 commits integrated in 3 months.

Developer *commitment* is studied here as an approximation of developer expertise, since the more a developer works on the Linux project or with a tool, the more expertise the developer may be assumed to acquire (on the Linux project and/or with the use of the tool). Figure 9 shows the distribution of commitment scores of developers for the different types of patches.

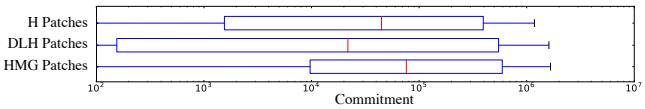


Figure 9: Commitment of developers Vs. Patch types.

DLH patches are shown to be produced by developers with a more varying degree of commitment (greater standard deviation). The median value of commitment is further lower than the median commitment for HMG patches. Finally, overall, the distributions of commitment values of developers indicate that H patch authors present lesser commitment than HMG patch authors.

We then use Spearman's ρ [43] to measure the degree of correlation between the commitment of developers and the number of tool-supported patches that they submit. We focus on specialized[17] developers of two very different kinds of code: mature file system (*fs*) code and early-development (*staging*) code. The correlation is then revealed to be higher ($\rho = 0.42$) for staging than for fs ($\rho = 0.11$). We also note that 64% of developers committing code in staging stick to this part of the code for over half of their contributions. Finally, developers specialized in *kernel* have never relied on tool support to produce a patch.

Bug detection tools are generally used by developers with (to some extent) knowledge of the code. Patch application tools, on the other hand, enable developers to remain committed to contributing patches to the code base.

4.4 Stability of Patches (RQ3)

Although patches are carefully validated before they are integrated to the mainline code base, a patch might be simply incorrect and thus the relevant code may require further changes, or the patch may simply be reverted. However, it is challenging to precisely detect and resolve such a change in recently patched code hunks. Even this requires heuristics that may prove to be error-prone. Thus, in this study, we focus on commits whose reverting is explicit.

It is common for software developers to cancel patches that they hastily committed to the code base. The `git revert` command is an excellent means for developers to roll back their commits. However, given the hierarchical organization in Linux, when a patch has reached the mainline, a simple revert (using git commands) is uncommon. The submitting developer (or another one) must write another patch explaining the need to revert. This patch again goes through the process to be accepted in the mainline. In this setting, the revert of a commit is likely strongly justified. We search for

[16] We considered only tools associated to at least 50 patches.

[17] speciality metric value greater than 50%

commits that are reverted by looking at commit messages where we have seen a pattern of the form "`revert <hash>`"[18].

We have found that 2.81% of H-patch commits have been later reverted. In contrast, only 0.27% and 0.32% respectively of DLH and HMG patch commits have been reverted. Figure 10 further provides the distributions of delays in reverting commits.

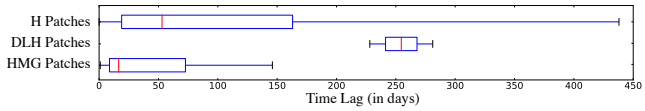


Figure 10: Time lag between patch integration and reverting.

H-patches revert delay distribution is the most spread. On average (median), a DLH patch, when it is reverted, will be so after 250 days (8 months). On the other hand, HMG patches will be reverted in less than a month (20 days). The median delay for revert is of 60 days for H patches.

> Tool-supported patches are generally stable. However, while patches fixing tool warnings may be found inadequate long after their integration, issues with patches generated based on fix patterns appear to be discovered quickly.

4.5 Bug Kinds (RQ4)

We study bug kinds in two dimensions: the spread of buggy code and the complexity of the bugs. We investigate the locality of patches as an approximation of the spread of buggy code, and the change operations at the level of Abstract syntax tree nodes modifications to approximate complexity of bugs.

4.5.1 Locality of Patches. The locality of patches is a key dimension for characterizing patches. Patch size has been measured in the literature [8, 38] in terms of the number of code locations that it involves, while several state-of-the-art automated repair approaches mostly focus on single/limited code changes to fix software. The Linux project is a particularly adequate study subject for this comparison since developers are often reminded that they must "solve a single problem per patch"[19]: fix operations are then generally separated from cosmetic changes.

A bug fix patch may involve changes across files. Figure 11 shows that most fixes are localized to a single file independently of the way they are constructed.

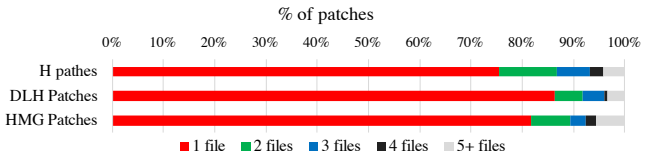


Figure 11: Distribution of patch sizes in terms of files.

DLH patches appear to be the more local, while more than 20% of H patches implement simultaneous changes in at least two files. Interestingly, we note that HMG patches include the largest proportion of patches (5.6%) that simultaneously change 5 files or more. Such patches are generated to fix pervasive bugs such as the wrong usage of an API, or to implement a collateral evolution.

[18] We use: `git show '+sha+' | grep -E -i "revert .[0-9a-f]5+ | commit .[0-9a-f]5+ | [0-9a-f]{40}$*`

[19] see `Documentation/SubmittingPatches`

We further investigate the locality of patches in terms of the number of code hunks (i.e., a contiguous group of code lines[20]) that are changed by a patch. Indeed, code files can be large, and a patch may variably spread changes inside the file, which, to some extent, may represent a degree of complexity of the fix. Figure 12 shows that H patches are more likely to involve several hunks of code than HMG and DLH patches.

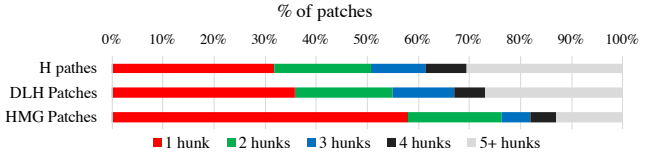


Figure 12: Distribution of patch sizes in terms of hunks.

Our observations on patch sizes suggest that developers, with or without bug finding tools, must correlate data and code statements across different code blocks to repair programs.

Finally, we compute the locality of the patches in terms of the number of lines that are affected by the changes. Such a study is relevant for estimating the proportions of isolated change (i.e., single-line changes) that fix bugs in the three scenarios of repairs. Figure 13 reveals that the large majority of patches that are manually crafted as responses to bug reports change several lines, with almost 70% patches impacting at least 5 lines. On the other hand, over 40% HMG patches impact only at most two lines of code.

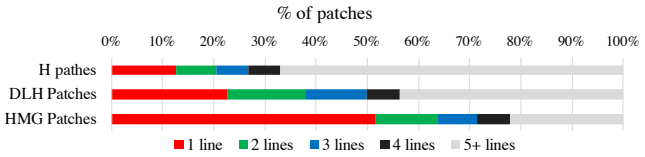
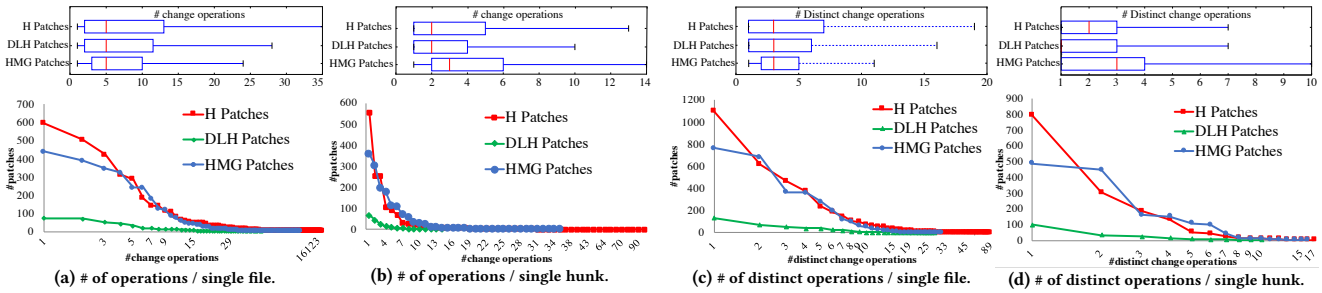


Figure 13: Distribution of patch sizes in terms of lines.

4.5.2 Change Operations in Patches. In general, line-based diff tools, such as the GNU Diff, are limited in the expression of the kinds of changes that can be identified since they consider only adds and removes, but no moves and updates [36]. Thus, to investigate change operations performed by patches, we rely on approaches that compute modifications based on abstract syntax trees (AST) [21]. Such approaches produce fine-grained results at the level of individual nodes. For this study, we consider an extended version of the open-source GumTree [14] with support for the C language [36]. This tool specifically takes into account additions, deletions, updates and moves of individual tree nodes, and has the goal of producing results that are easier for users to understand than those of GNU Diff.

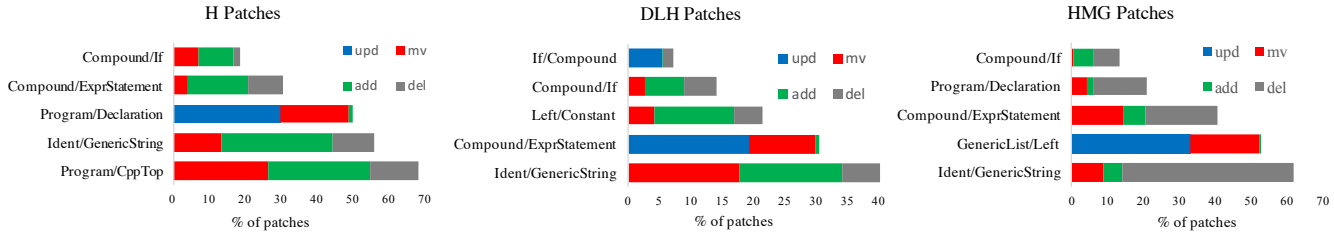
The output of GumTree is an edit script enumerating a sequence of operations that must be carried out on an AST tree to yield the other tree. To that end, GumTree implements a mapping algorithm between the nodes in two abstract syntax trees. This algorithm is inspired by the way developers manually look at changes between two files, first searching for the largest unmodified chunks of code (i.e., isomorphic subtrees) and then identifying modifications (i.e., given two mapped nodes, find descendants that share a large percentage of common mappings, and so on). Given those mappings, GumTree leverages an optimal and quadratic algorithm [11] to

[20] https://www.gnu.org/software/diffutils/manual/html$_n$ode/Hunks.html



(a) # of operations / single file. (b) # of operations / single hunk. (c) # of distinct operations / single file. (d) # of distinct operations / single hunk.

Figure 14: Distribution of change operations (Total # of operations & # of distinct operations in patches).



(a) H patches. (b) DLH patches. (c) HMG patches.

Figure 15: Top-5 change operations appearing at least once in a patch from the three processes.

compute the edit script. More details on the algorithm can be found in the original articles [11, 14].

For simplicity, in this paper, we express change operations in their abstract form as a triplet "*scope/element:action*" where *scope* represents the type of node (e.g., the program, an `If` block, a compound block, a generic list, an identifier, etc.) where the change occurs, *element* represents the element (e.g., an expression, a declaration, a generic string, a compound block, an if block, etc.) that is changed and *action* represents the move/update/add/delete operators that are used. This abstract representation indeed does not take into account any variable names and functions involved (and available in the output of GumTree). Figure 16 shows a patch example for a change operation where a new `If` block code is inserted.

```
diff --git a/drivers/gpu/drm/i915/intel_display.c b/.../drm/i915/intel_display.c
index 6e0d828..182f849 100644

--- a/drivers/gpu/drm/i915/intel_display.c
+++ b/drivers/gpu/drm/i915/intel_display.c
@@ -13351,6 +13351,9 @@ int intel_atomic_prepare_commit(struct drm_device *dev,
       for_each_crtc_in_state(state, crtc, crtc_state, i) {
+ if (state->legacy_cursor_update)
+ continue;
+
                ret = intel_crtc_wait_for_pending_flips(crtc);
```

Figure 16: Example of `Compound/If:add` – Add an `If` block.

Figure 14 illustrates the distributions of the number of operations that are performed in a patch. To limit the bias of changes that are identically performed in several files (e.g., Coccinelle collateral evolutions), we focus on patches that touch a single file, then on patches that are limited to a single hunk. All distributions are long-tail, revealing that most patches apply very few operations in terms of number and variety. While the three processes have similar average (median) values of change operations performed on a file, HMG patches appear to implement changes with a consistent number of operations (limited standard deviation). On the other hand, when we consider change operations at the hunk level, DLH patches apply fewer operations than HMG patches[21].

[21] We have checked with MWW tests that the difference is statistically significant.

Figure 15 summarizes the top-5 change operations that are recurrently implemented by patches constructed in the different processes considered in our study. Changes performed appear to be specific for each process. For example, while `Ident/GenericString` and `Compound/If`-related change operations occur in most patches, they do not display the same proportions in terms of additions, moves, updates and deletions.

> Overall, patches, following their construction process, differ in terms of size (i.e., the spread of the buggy code that they repair) and in the nature of change operations that they implement (i.e., the complexity of the bug).

5 DISCUSSIONS

We discuss the implications of our findings for the software engineering research community, in particular, the automated research field, and enumerate the threats to validity that this study carries.

5.1 Implications

As the field of automated repair is getting mature, the community has started to reflect (i) on whether to build human-acceptable or readable patches [20, 32], (ii) on the suitability of automated repair fixes [41], (iii) on the relevance of patches produced by repair tools [51]. Our work continues this reflection from the perspective of the acceptance of tool-support in patch construction. We further acknowledge that HMG patches considered in this study are not constructed in the same spirit as in automated repair: indeed, automated repair approaches make no a-priori assumption on what and where the fault is, while tools such as Coccinelle [9] produce patches based on fix patterns that match buggy code locations. Nevertheless, given the lack of integration of automated repair in a real-world development process, we claim that investigating Linux patch cases can offer insights which can be leveraged by the research community to understand how the developer community

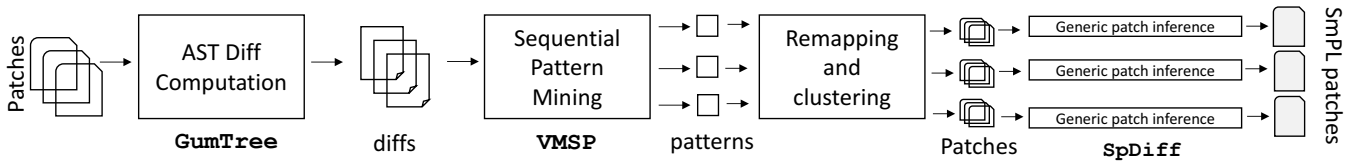


Figure 17: Searching for redundancies among patches that fix warnings of bug finding tools (i.e., DLH patches).

can accept tool-supported patches, and the automation of what kind of fixes can be readily accepted in the community.

On manual Vs. tool-supported patches. As illustrated in Section 4.1, tool-supported patch construction is becoming frequently and widely used in the Linux Kernel development. In particular, HMG patches account for a larger portion of recent program changes than H patches. This suggests that both (1) developers gradually accept to use patch application tools such as Coccinelle [9] since they are effective to automatically change similar code fragments and (2) there are many (micro) code clones [46] in the code base. Regarding spatial distribution, DLH and HMG patches are committed to 'staging' (22-47%) while H patches in 'staging' account for only 1%. This may indicate that experimental features have more opportunities for tools to help write bug fixing patches. It implies indeed that, for early development code, the community almost exclusively relies upon tools to solve common bugs (e.g., in relation with programming rules, styles, code hardening, etc.) by novice programmers (i.e., not necessary specialized in kernel code), before expert developers can take over. Thus, reliable automated repair techniques could be beneficial in a production development chain as debugging aids. This finding comforts the human study recently conducted by Tao et al. [45] which suggested that automated repair tools can significantly help debugging tasks.

On the delay in patch acceptance. We have observed a delay in the acceptance of tool-supported patches by maintainers. However, given the differences in change operations with fully manual patches, it is likely the case that tool-supported patches are fixing less severe bugs, which makes their integration a less crucial issue for maintainers.

Furthermore, negative percentages in evolution gap between submission and acceptance (cf. Figure 7) suggests that there are many HMG patches that are integrated into the mainline code base without being discussed by maintainers. This finding implies that once the fix pattern has been validated, patches appear to be accepted systematically.

On the nature of bugs being fixed. The study of patch locality shows results that are in line with a previous study [51] which revealed that most fix patches only change a single file. Nevertheless, we have found that, in practice, even tool-supported patches, in a large majority, modify several lines to fix warnings by bug detection tools (which, by the way, generally flag a single line in the code). Although patch size does not, by any means, imply ease of realization, our results suggest that there are considerable numbers of repair targets and shapes that automated repair should aim for.

It is also noteworthy that the spread of change operations over several files may carry different implications for the patch construction processes. For example, while a coccinelle patch may be applying the same change pattern over several files to fix an API function usage, a human patch modifying several files may actually carry data and behavior dependencies among the changes.

5.2 Exploiting Patch Redundancies

A large body of the literature on program repair has discussed findings on the repetitiveness/redundancy of code changes in real-world software development [3, 34]. Unfortunately, such findings are not readily actionable in the context of automated repair since they do not come with insights on how such redundant patches will be leveraged in practice. Indeed, although it is possible to abstract redundant patches to recommend bug fix actions [5], only a few research directions manage to contextualize them, to some extent, for repair scenarios [26]. Actually, researchers discuss such redundancies for enriching the repair space with change operations that are more likely to be appropriate fix operations.

With this study, we see concrete opportunities for exploiting patch redundancies for systematically building patches and applying (or recommending) them to a specific identified and localized buggy piece of code. Indeed, bug detection tools, which are used by various developers who then craft fixes based on specific warnings, and patch application tools, which are based on fix patterns, can be leveraged in an automated repair chain. The former will be used in the bug detection and localization steps while the latter will focus on building concrete patches based on patterns found in a database of human fixes created to address warnings by bug detection tools.

To demonstrate the feasibility of this research direction, we have conducted a study for searching redundancies in patches constructed following warnings by bug detection tools, and investigating the possibility of producing a generic patch which could have been used to derive these concrete patches. Nevertheless, although generic patch inference has been a very fertile research direction in the past [1, 2, 30, 31], we have experimented available tool supports and found that they do not scale in practice. We have thus devised a process to split the set of patches into clusters, each containing patches presenting similar change operations. Figure 17 depicts the overall process. Based on GumTree sequences of change operations, we rely on a sequential pattern mining tool to extract maximal sequential patterns. We use a fast implementation of VMSP [15] to find recurrent change patterns at the level of the abstract change operations expressed in Section 4.5. Then, we build clusters of patches based on the elicited patterns, and leverage SpDiff [1] to attempt the inference of a unique SmPL patch which could instantiate the common redundant concrete repair actions performed in the patches.

With this process, starting with a set of 571 DLH patches, we were able to build 37 clusters based on change operations patterns. Among the clusters, 10 led to the generation of a common generic patch. We then manually investigated the commit messages associated with the patches in clusters that produced a generic patch, and found that they indeed largely dealt with the same bug type. This final check confirms, to some extent, the potential to collect fix patterns from human repair processes to build an automated repair chain leveraging bug detection tools.

5.3 Threats to Validity

We have identified the following threats to validity to our study:
External validity – We focus on Linux only. It is, however, one of the largest development project, one of the most diverse in terms of developer population, with a significant history for observing trends, and implementing strict patch submission guidelines that try to systematize the tracking of change information. To the best of our knowledge, Linux is the best candidate for observing various patch construction processes, as it encourages the use of tools for bug detection and patching.
Construct validity – We rely on a number of heuristics to collect and process our datasets. We have nevertheless, by design, chosen to be conservative in the way we collect patches in each process with the objective of having reliable and distinctive sets for each process, to further enable replication.
Internal validity – The metrics that we leverage to elicit the differences among the different processes may lead to biased results. However, those metrics were also used in the literature.

6 RELATED WORK

6.1 Program Repair

6.1.1 Studies on Human-Generated Patches. Studies on patches, generated by human developers, focus on investigating existing patches fully written by developers (i.e., H patches) rather than devising a new technique. Pan et al. explored syntactic bug fix patterns in seven Java projects [39]. This study extracted 27 bug fix patterns. Martinez and Monperrus identified common program repair actions (patterns) [27], and Zhong and Su reported statistics on 9,000 real bug fixing patches collected from Java open source projects [51]. These studies examined features of real bug fixes against whether automated repair techniques can be applied to fix those bugs. In addition, Barr et al. formulated a hypothesis called "*plastic surgery hypothesis*" [3]. They studied how many changes can be graftable by using snippets that can be found in the same code base where the changes are made.

6.1.2 Studies on Tool-aided Patches. As discussed in Sections 1 and 3, generating tool-aided patches indicates that developers create program patches with an aid of tools, rather than generating patches from scratch. Tao et al. supposed that automated repair tools can provide aids to debugging tasks [45]. They adopted Par [20] as a patch recommendation tool and gave patches generated by the tool to experiment participants. The findings include that automatically generated patches can significantly help debugging tasks. MintHint [18] is a semi-automatic repair technique, which can help developer find correct patches. This technique does statistical correlation analysis to locate program expressions likely to perform repaired program executions.

6.1.3 Automated Patch Generation. Generating patches with automated tools implies minimizing a developer's effort in debugging. It often indicates that fully automated procedures including fault localization, code modification, and patch verification. Recent endeavors achieved an impressive progress as follows.

Weimer et al. [48] proposed GenProg, an automatic patch generation technique based on genetic programming [22]. This technique randomly mutates buggy statements to generate several different program variants that are potential patch candidates. In 2012, the authors extended their previous work by adding a new mutation operation, replacement and removing the switch operation [24]. SemFix [33] leverages program synthesis to generate patches. The technique assumes that buggy predicates are an unknown function to be synthesized. The technique is successful for several bugs, but it is only applicable to "one-line bug", in which only one predicate is buggy. DirectFix [28] and Angelix [29] extended Semfix so that it can generate patches for bugs in larger and complex (w.r.t the search space) programs in a simpler way. PAR [20] automatically generates patches by using fix patterns learned from human-written patches. This technique is inspired by the fact that patches are redundant.

6.2 Patch Acceptability

Fry et al. conducted a human study to indirectly measure the quality of patches generated by GenProg by measuring patch maintainability [16]. They presented patches to participants and asked maintainability related questions developed by Sillito et al. [40]. They found that machine-generated patches [24] with machine-generated documents [10] are comparable to human-written patches in terms of maintainability. Par [20] is presented to deal with nonsensical patches. The approach generates patches based on fix patterns, which are learned from human-written patches. The fix patterns generalize common repair actions from more than 60,000 real bug fixes enabling Par to avoid generating nonsensical patches.

6.3 Program Matching and Transformation

SYDIT [30] automatically extracts an edit script from a program change. In its scenario, a user must specify the program change to extract the edit script from. Coccinelle [9], on the other hand, directly lets the user specify the edit script in a user-friendly language, and performs the transformation by matching the change pattern with code context. It has been used in several debugging tasks in the literature [4–7, 37]. LASE [31] differs from SYDIT as it can generate a generalized edit script based on multiple changes of Java programs. Another approach in this direction is SpDiff [1, 2] supports the extraction of a subset of common changes (i.e., SmPL patches that are fed to Coccinelle) from several concrete patches.

7 CONCLUSION

We have studied the impact of tool support in patch construction, leveraging real-world patching processes in the Linux kernel development project. We investigated the acceptance of tool-supported patches in the development chain as well as the differences that may exist in the kinds of bugs that such patches fix in comparison with traditional all-hand written patches. We show that in the Linux ecosystem, bug detection and patch application tools are already heavily used to unburden developers, and already enable relatively complex repair schema, contrasting with a number of repair approaches in the state-of-the-art literature of automated repair. An artefact dataset on this study is available at https://goo.gl/f1mRMM.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOMMEND C15/IS/10449467 and FIXPATTERN C15/IS/9964569.

REFERENCES

[1] Jesper Andersen and Julia L. Lawall. 2008. Generic Patch Inference. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA, 337–346.

[2] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. 2012. Semantic Patch Inference. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA, 382–385.

[3] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 306–317.

[4] Tegawende Bissyande. 2013. *Contributions for improving debugging of kernel-level services in a monolithic operating system.* Ph.D. Dissertation. Université Sciences et Technologies-Bordeaux I.

[5] Tegawendé F Bissyandé. 2015. Harvesting Fix Hints in the History of Bugs. *arXiv preprint arXiv:1507.05742* (2015).

[6] Tegawendé F Bissyandé, Laurent Réveillère, Julia L Lawall, and Gilles Muller. 2012. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 60–69.

[7] Tegawendé F Bissyandé, Laurent Réveillère, Julia L Lawall, and Gilles Muller. 2014. Ahead of time static analysis for automatic generation of debugging interfaces to the Linux kernel. *Automated Software Engineering* (2014), 1–39.

[8] Tegawende F. Bissyande, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, and Laurent Reveillere. 2013. Empirical Evaluation of Bug Linking. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Washington, DC, USA, 89–98.

[9] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 2009. A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Savannah, GA, USA, 114–126.

[10] Raymond P.L. Buse and Westley R. Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, New York, NY, USA, 33–42.

[11] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 493–504.

[12] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 73–88.

[13] Linus Torvalds et al. Last Accessed: Feb. 2017. GIT. http://git-scm.com/. (Last Accessed: Feb. 2017).

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 313–324.

[15] Philippe Fournier-Viger, Cheng-Wei Wu, Antonio Gomariz, and Vincent S. Tseng. 2014. VMSP: Efficient Vertical Mining of Maximal Sequential Patterns. In *Advances in Artificial Intelligence*. Springer, Cham, 83–94.

[16] Zachary P. Fry, Bryan Landau, and Westley Weimer. 2012. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 177–187.

[17] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485–501.

[18] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. MintHint: Automated Synthesis of Repair Hints. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, 266–276.

[19] Linux Kernel. Last Accessed: Feb. 2017. Bugzilla Tracking System. https://bugzilla.kernel.org. (Last Accessed: Feb. 2017).

[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 802–811.

[21] Miryung Kim and David Notkin. 2006. Program Element Matching for Multi-version Program Analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*. ACM, New York, NY, USA, 58–64.

[22] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (1 ed.). The MIT Press.

[23] Julia L. Lawall, Julien Brunel, Nicolas Palix, Rene R. Hansen, Henrik Stuart, and Gilles Muller. 2009. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 43–52.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *Software Engineering, IEEE Transactions on* 38, 1 (Feb. 2012), 54 –72.

[25] LIP6. Last Accessed: Feb. 2017. Coccinelle. http://coccinelle.lip6.fr/. (Last Accessed: Feb. 2017).

[26] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, New York, NY, USA, 702–713.

[27] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (Feb. 2015), 176–205.

[28] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 448–458.

[29] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, New York, NY, USA, 691–701.

[30] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. SYDIT: creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, New York, NY, USA, 440–443.

[31] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 502–511.

[32] Martin Monperrus. 2014. A Critical Review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, 234–242.

[33] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 772–781.

[34] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ACM, New York, NY, USA, 315–324.

[35] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys'08: Proceedings of the 2008 ACM SIGOPS/EuroSys European Conference on Computer Systems*. Glasgow, Scotland, 247–260.

[36] Nicolas Palix, Jean-Rémy Falleri, and Julia Lawall. 2015. Improving pattern tracking with a language-aware tree differencing algorithm. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 43–52.

[37] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. Newport Beach, California, USA, 305–318.

[38] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. 2014. Faults in Linux 2.6. *ACM Trans. Comput. Syst.* 32, 2 (June 2014), 4:1–4:40.

[39] Kai Pan, Sunghun Kim, and E. James Whitehead. 2008. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (Aug. 2008), 286–315.

[40] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, New York, NY, USA, 23–34.

[41] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 532–543.

[42] Jasper Spaans. Last Accessed: Feb. 2017. The Linux Kernel Mailing List. (Last Accessed: Feb. 2017). http://lkml.org.

[43] C. Spearman. 1904. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 15, 1 (1904), 72–101.

[44] Synopsys. Last Accessed: Feb. 2017. Coverity. http://www.coverity.com/. (Last Accessed: Feb. 2017).

[45] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 64–74.

[46] R. van Tonder and C. Le Goues. 2016. Defending against the attack of the micro-clones. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4.

[47] Linus Torvalds. Last Accessed: Feb. 2017. Linux kernel git tree. http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/. (Last Accessed: Feb. 2017).

[48] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 364–374. ACM ID: 1555051.

[49] Sparse Wiki. Last Accessed: Feb. 2017. Sparse. https://sparse.wiki.kernel.org. (Last Accessed: Feb. 2017).

[50] Wikipedia. Last Accessed: Feb. 2017. Benevolent dictator for life. http://en.wikipedia.org/wiki/Benevolent$_{d}$ictator$_{f}$or$_{l}$ife. (Last Accessed: Feb. 2017).

[51] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press, Piscataway, NJ, USA, 913–923.