The Thesis Committee for Jinru Hua
certifies that this is the approved version of the following thesis:

A Case Study of Cross-Branch Porting in Linux Kernel

APPROVED BY

SUPERVISING COMMITTEE:

Miryung Kim, Supervisor

Herb Krasner

A Case Study of Cross-Branch Porting in Linux Kernel

by

Jinru Hua, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

A Case Study of Cross-Branch Porting in Linux Kernel

Jinru Hua, M.S.E
The University of Texas at Austin, 2014

Supervisor: Miryung Kim

To meet different requirements for different stakeholders, branches are widely used to maintain multiple product variants simultaneously. For example, Linux Kernel has a main development branch, known as the mainline; 35 branches to maintain older product versions which are called stable branches; and hundreds of branches for experimental features. To maintain multiple branch-based product variants in parallel, developers often port new features or bug-fixes from one branch to another. In particular, the process of propagating bug-fixes or feature additions to an older version is commonly called *backporting*.

Prior to our study, backporting practices in large scale projects have not been systematically studied. This lack of empirical knowledge makes it difficult to improve the current backporting process in the industry. We hypothesize that cross-branch porting practice is frequent, repetitive, and error-prone. It

requires significant effort for developers to select patches that need to be backported and then apply them to the target implementation. We carried out two complementary studies to examine this hypothesis.

To investigate the extent and effort of porting practice, this thesis first describes a quantitative study of backporting activities in Linux Kernel with a total of 8 years version history using the data of the main branch and the 35 stable branches. Our study shows that backporting happenes at a rate of 149 changes per month, and it takes 51 days to propagate patches on average. 40% of changes in the stable branches are ported from the mainline and 64% of ported patches propagate to more than one branch. Out of all backporting changes from the mainline to stable branches, 97.5% are applied without any manual modifications. To understand how Linux Kernel developers keep up to date with development activities across different branches, we carried out an online survey with engineers who may have ported code from the mainline to stable branches based on our prior analysis of Linux Kernel version history. We received 14 complete responses. The participants have 12.6 years of Linux development experience on average and are either maintainers or experts of Linux Kernel.

The survey showes that most backporting work is done by the maintainers who know the program quite well. Those experienced maintainers can easily identify the edits that need to be ported and propagate them with all relevant changes to ensure consistency in multiple branches. Inexperience developers are seldom given an opportunity to backport features or bug-fixes to

stable branches.

In summary, based on the version history study and the online survey, we conclude that cross-branch porting is frequent, periodic, and repetitive. It requires a manual effort to selectively identify the changes that need to be ported, to analyze the dependency of the selected changes, and to apply all required changes to ensure consistency. To eliminate human's omission mistakes, most backporting work is done only by experienced maintainers who can identify all relevant changes along with the change that needed to be backported. Currently inexperienced developers are excluded from cross-branch porting activities from the mainline to stable branches in Linux Kernel.

Our results call for an automated approach to identify the patches that require to be ported, to collect context information to help developers become aware of relevant changes, and to notify pertinent developers who may be responsible for the corresponding porting events.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Large systems often use multiple development branches to maintain multiple product variants for different stakeholders. Linux Kernel is a prominent example of large open source software that has hundreds of branches evolving simultaneously, thousands of developers collaborating together, and millions of end users using the Linux operating system and its derived products. The main development branch in Linux Kernel is called the *mainline*. The mainline incorporates all kinds of changes, both the latest features and bug fixes. Not all of these changes are fully tested before the new version is released. Therefore, Linux Kernel maintains a separate set of stable branches for users who simply want the security and bug fixes, but not a whole new version. To maintain these stable branches, developers often need to propagate bug fixes and features from one version to another, and this process is known as *backporting*. Here *backporting* means applying software modifications made in new versions to older versions [34] or to merge a commit from the mainline to maintenance branches [37]. Apart from Linux Kernel, backporting is also commonly used in a number of product versions, such as Red Hat, Fedora, and FreeMind [34]. For another example, many features of Windows Vista

were backported to Windows XP when the Service Pack 3 was released for Windows XP to facilitate compatibility of applications [43].

However, there has been a lack of study on the advantages and disadvantages of the backporting process. We hypothesize that backporting is error-prone and time-consuming for the following reasons. First, a backporting practice requires manual effort to identify the patches that need to be ported and apply similar changes to peer contexts. Second, porting changes across branches can be error-prone when developers do not consider the dependencies and constraints in the target branch carefully. Lastly, after Linux developers locate all patches that need to be ported, they are required to take the responsibility to make sure that the ported change is consistent across all stable branches.

To investigate the extent and repetitive effort of backporting practice in Linux Kernel, we conducted a version history study followed by a survey. We describe our approach and study results in this thesis.

1.2 Background

Many companies maintained collections of similar products as software product lines by cloning and adapting artifacts of existing product variants. Transforming such cloned product variants into a "single copy" software product line representation was considered an important software re-engineering activity with numerous tools and methodologies [17, 8, 4]. However, to save time and reduce cost, the common practice in industry was not to migrate

similar products into a software product line, but to maintain multiple cloned products and modified them to fit new requirements [3, 32, 11].

Different from these approaches that modified the cloned product variants in software product lines, we considered each branch as an individual version of the same product and investigated a common porting practice in software maintenance—backporting, to understand the change propagation from the mainline to multiple maintenance versions. In particular, we targeted a large scale open source project with around 16 million lines of code in a low level language C, which was not supported by any of the approaches above.

Other studies in branching investigated how branches supported collaborative development [2, 24, 1] and how to eliminate conflicts for potential integration failures [10, 5]. Rather than regarding branches as subsystems or subsets of the product, we considered the mainline and stable branches as a complete product in different versions. Instead of focusing on eliminating conflicts in collaborative development, we investigated similar cross-branch modifications and porting features in the `git` version control system.

Current similar changes identification studies [25, 19, 21] focused on general software modifications. On the other hand, we studied a typical porting practice in software maintenance—backporting, which was mainly for bug-fixes, not feature additions. The study of recurring bug-fixes [23] was similar to our work. However, our study investigated cross-branch patch applications and analyzed porting practice from developers' perspectives. We also inves-

tigated how developers identified the patches that should be backported and which information was useful for making the decision to backport. None of these questions has been answered by preceding research.

Compared to the studies on clone detection in large scale projects [20, 22, 31], which mainly focused on the porting from one project to another at a release level, we performed a more fine-grained analysis at a commit level to investigate backporting practice from the main development branch to multiple maintenance branches. Apart from considering similar code segments, we also identified similar edit operations (insert and delete) when studying backporting activities using REPERTOIRE [25]. We identified 11774 backporting patch pairs out of 390,581 patches in 8 years' development of Linux Kernel and evaluated the precision and recall of this backporting dataset against a ground truth dataset from *bugzilla*.

1.3 Research Questions

The purpose of this thesis is to investigate the cross-branch porting activities in Linux Kernel. We selected Linux Kernel because it was one of the most typical open source projects with multiple branches in parallel. Due to its fast updating process, not all new features can be fully tested, which leads to frequent porting practices of bug-fixes across branches. Its large developer community which consists of 5,000 to 6,000 developers also provides a great resource for our user study via online survey.

Most development activities in Linux Kernel happen in the main branch,

which is known as the mainline. Apart from it, Linux Kernel also maintains a set of "stable" branches for people who simply prefer bug fixes, but not a whole new version. Bug-fixes are frequently ported from the mainline to stable branches, and this process is known as *backporting*.

In this thesis, we focus on the following questions:

RQ 1 What percentage of patches in stable branches are coming from the mainline?

RQ 2 How long does it take for a patch to propagate from the mainline to stable branches in Linux Kernel?

RQ 3 What percentage of backporting patches are directly applied without any manual adaptation?

RQ 4 How much repetitive effort is involved in porting a patch from the mainline to multiple stable branches?

RQ 5 What risks and challenges are associated with backporting changes from the main branch to other branches?

RQ 6 How do developers currently determine a patch that must be ported from the mainline to other branches?

We investigated the version history of Linux Kernel to understand the extent and repetitive effort of backporting, and answer the questions from RQ 1 to RQ4 in Section 3. In order to answer RQ 5 and 6, we conducted a survey

with the developers who might have ported code across branches based on our backporting analysis of the version history.

1.4 A Version History Analysis of Backporting in Linux Kernel

In order to comprehensively characterize backporting practices in Linux Kernel, we investigated a total of 8 years version history data of the main branch and 35 maintenance branches using REPERTOIRE [25]. First, by adapting REPERTOIRE from `CVS` to `git` version control system, we refined REPERTOIRE from release-level to commit-level for a more fine-grained similar change identification. We identified 369 similar patch pairs using REPERTOIRE by comparing both the content and edit operations in the patches. Next, we traced special `git` porting-support commands—`cherry-pick` and `rebase`. `cherry-pick` command simply applies a patch from one branch to another; while `rebase` takes all the changes committed on one branch and replays them on another branch. We located 11448 backporting patch pairs with this method. Our accuracy evaluation on 347 resolved bug reports from the Linux Kernel bug repository *bugzilla* indicated that our prototype detected cross-branch backporting events with the precision of 89.1% and recall of 75.2%.

The following paragraphs summarize our results of the version history study of backporting in Linux Kernel.

- Approximately 40% of all patches in 35 stable branches came from the

mainline. The average porting time from the mainline to stable branches was 51 days. Manually applied backporting patches took longer, 74 days on average.

- While most patches were ported to stable maintenance branches without any adaptation, some were manually edited to fit the target branch context. These manually applied backporting patches had contents that were about 60% similar to the corresponding reference patches.

- 64% of ported patches were propagated to more than one branch, indicating that there was a significant redundancy of backporting effort. In an extreme case, a single patch from the mainline was ported to 14 stable branches at maximum.

These results indicate that backporting is frequent, periodic, and repetitive. We hypothesize that backporting requires a manual effort of selecting and adapting patches to the target implementation, which we investigate further in the next section.

1.5 A Survey on Cross-Branch Porting Practice in Linux Kernel

To examine the challenges, risks, and repetitive effort of backporting practices in Linux Kernel, we conducted an online survey with engineers who might have ported code from the mainline to stable branches based on our prior analysis of Linux Kernel version history. The survey consisted of three

parts, (1) how developers identify a patch that should be backported, (2) how developers adapt the backporting patch to the target context with all relevant changes and (3) how the existing tools support backporting activities.

We received 22 responses out of 228 developers of Linux Kernel, 14 of them completed all 15 questions. Our participants had more than 20 years software development experience with 12.6 years experience on Linux development on average. Most of them were subsystem maintainers or Linux Kernel experts. We summarize our results of the survey below:

- Following the development process of Linux Kernel, only serious bug-fixes were backported to stable branches. Developers made the decision of backporting intuitively based on their experience and understanding of the mainline and stable branches.

- Experienced developers held diverging perspectives on the difficulties of backporting practice. One third regarded identifying backporting patches as hard, while one third believed it was easy. The rest held a neutral opinion on the backporting patch identification. We received almost the same diverging result when we asked whether they regarded backporting as risky and error-prone.

- Aligned with our quantitative analysis, 85% of developers reported that during the process of applying mainline patches to stable branches, most modifications were trivial, such as renaming and simple rewriting of the functions which could be done within one hour or so.

- All participants strongly believed that change descriptions played an important role to identify the patches that must be backported. 63.64% of participants believed that the *diff* content of the patch was also useful to decide whether a patch was relevant to other branches and should be ported.

- Almost all participants subscribed to the mailing lists to receive development activities in the mainline. Some of them also discussed personally with fellow developers to know about mainline development activities.

1.6 Summary

This thesis presents two complementary studies to examine whether cross-branch porting practice is frequent, repetitive, and error-prone which requires manual effort to select and adopt the changes that need to be ported.

We first conducted a quantitative study using the version history data of Linux Kernel, and found that the maintenance effort of cross-branch porting was significant. Though most changes could propagate to the maintenance branches with little effort, we found it was time-consuming to identify all relevant changes that needed to be ported along with the backported patches to ensure the change consistency across branches.

To examine whether our version history study matched developers' perception about cross-branch porting in practice, we performed a follow-up survey with Linux Kernel developers and found that non-experts were excluded

from the backporting tasks due to a lack of fully comprehension of the dependency and constraints in target branches.

In order to reduce repetitive effort and allow non-experienced developers to take the responsibility of backporting, we believe that automated tool support for the backporting can be useful to identify the changes that need to be ported and apply the changes from the master branch to maintenance branches.

1.7 Thesis Outline

Chapter 2 presents related work on software product lines, branch-based collaboration, similar changes identification and clone detection. Chapter 3 describes our version history analysis of backporting in Linux Kernel and Chapter 4 presents the follow-up survey with the developers that may have ported code from the mainline to stable branches based on the version history study. We conclude our thesis with Chapter 5 along with the discussion of threats to validity and future work.

Chapter 2

Related Work

2.1 Cloned Variant Management in Software Product Line Development

It is common to maintain multiple product variants as a software product line to meet different requirements from different users. Developers adopt a *clone-and-own* approach to clone and modify the existing product to fit for the requirements. To understand cloning activities in collections of similar products in industrial product lines, Dubinsky et al. [7] conducted an exploratory study and found that it was difficult to make sure that changes and bug-fixes in one product were propagated to all required products correctly, and developers must perform repetitive tasks for each product variant. Other works in software product line attempted to identify common variants or similar functions which could be moved to the core [17, 11, 32]. They refined cloned software products and migrated multiple product instances to a principled product line [3, 8, 4].

However, our research differs from the preceding research on the software product line in the following two aspects. First, instead of focusing on industrial application product lines, we focused on a single large scale open source project—Linux Kernel to study the porting activities written in a low-

level language C. Second, instead of analyzing all cloning activities between similar product variants, we investigated the porting activities only from the main development branch to multiple maintenance branches.

Considering that developers prefer to clone the existing products and modify them to fit new requirements, Rubin et.al [28] took a systematic top-down approach to identify a set of clone operators. They broke the activities down into individual clone management operators and showed that these operators supported the case when existing clone variants were maintained as is [29, 27].

Yet their approach could only cover a subset of activities related to the development, which could hardly fit into large scale systems with various modification activities. Our work used a token-based clone detection tool for large scale systems—*CCFinder* [14]. We successfully investigated porting activities with a data set of 390,581 patches and identified 11,774 patch pairs for a large scale open source project with 15.8 millions lines of code in C.

2.2 Branch-Based Collaboration

Software development for large projects is often a collaborative and team-based enterprise. To isolate changes and manage the complexity of coordination work in parallel, large software projects use branches to decompose the teams and the tasks [24]. Bird et al. [2] claimed that the files in a single branch were evolved cohesively and developers who worked on the same branch represented a virtual team in both technical and organizational prospectives.

They further assessed the usefulness of individual branches using a *what-if* analysis and identified *high-cost-low-benefit* branches in Windows for a possible branch removal [1].

Our research differs from their analysis, since we consider each branch as a version of the system as a whole rather than a subset of the system with different modules. We analyzed porting activities based on the idea that the mainline and stable branches were different versions of Linux Kernel, and each branch individually represented all required modules.

Other studies on branching argued that branches might introduce additional overhead to move code across branches. Shihab et al. [30] mentioned that this overhead could lead to unexpected dependencies and conflicts which resulted in potential integration failures. Brun et al. [5] found that conflicts were frequent and persistent in branch-based collaborative development, as developers had inconsistent copies of a shared project. To avoid textual, build, and testing conflicts, they implemented the tool *Crystal* to proactively detect pending conflicts and provide concrete advice. Aligned with *Crystal*, *WeCode* [10] continuously merged committed and uncommitted changes to detect conflicts and notified developers for the potential resolution of conflicts.

Rather than eliminating conflicts in collaborative development, our study identified similar modifications across different branches, and investigated porting features in a `git` version control system, such as `cherry-pick` and `rebase`, which we describe in Section 2.5.

2.3 Similar Changes Identification

Developers often make similar changes to multiple places. Ray et al. [25] that 14% of the edits were ported across forked projects—FreeBSD, OpenBSD, and NetBSD; 40% of active developers were involved in porting patches, and more than 50% of ported edits propagated within one year. By investigating the time required for porting changes from one project to another, their study result showed that porting practice seemed to heavily depend on developer doing their porting job on time, which required an automated approach of applying similar program transformations to peer contexts.

Apart from investigating the extent and developers who involved themselves in the porting, we investigated the repetitive effort involved in porting changes from the mainline to multiple stable branches and conducted a complementary survey with Linux developers to understand porting practices from developers' perspectives.

Other systematic editing tools automatically made similar changes across different locations. SYDIT [18] generated an edit script from a single systematic edit example and required developers to specify the target locations before applying the transformation. LASE [19] was able to identify all edit locations and transform the code based on the edit script. Yet both approaches require developers to demonstrate one or more examples to generate scripts.

Negara et al. [21] mined fine-grained code transformation sequences at an AST level to understand repetitive code change patterns. Nguyen et al. [23]

considered the repetitive bug-fixes with slight modifications on multiple code fragments to one or more revisions, which was known as *recurring bug-fixes*. Using a graph-based representation of object usages, they identified code peers, recognized recurring bug-fixes, and recommended changes for code units from the bug fixes of their peers.

In contrast to their approaches that compared AST nodes and corresponding edit operations in a high-level language Java, we used a token-based clone detection tool—CCFinder [14] to extend the scalability of similar change identifications in large scale systems written in a low-level language C.

2.4 Clone Detection in Large Scale Projects

Prior research showed that many large scale systems had a large amount of common code. For example, Gabel et al. [20] investigated 6000 software projects with over 420 million lines of code and found a common syntactic redundancy at the level of one to seven lines, Livieri [31] studied 136 versions of the Linux Kernel and found a coarse-grain backporting trends from parallel development branches while Cordy [6] found evidence of ported code in device driver modules between Linux and FreeBSD. Nguyen et al. [22] also showed that cross-project porting was more repetitive and stable than within-project code clone and MCIDiff [16] compared multiple clone instances simultaneously which aimed to summarize syntactic, semantic, and differential patterns in code clone.

We analyzed not only similar code but also similar changes. We con-

ducted a more fine-grained analysis to identify porting in the patch level and investigated the porting-support features provided by the `git` version control system.

Other analyses of clone detection concentrated on the consequence of faulty adaptation process and inconsistent updates. CP-Miner [15] was one of the first tools in detecting porting errors. With CP-Miner, Li et al. found that developers often adapted ported code in the target implementation—at least one identifier was renamed in 65% of the ported code, and in 27% cases at least one statement was inserted, modified, or deleted. Jiang et al. [12] presented evidence of porting errors when similar code appeared in different contexts. Juergens et. al [13] created a tool to identify inconsistent clone with context awareness. DejaVu [9] was another scalable system for detecting these general syntactic inconsistency bugs based on the assumption that duplicated code was generally intended to remain identical. Detecting and characterizing porting inconsistencies with a state-control and data-dependence analysis technique, SPA [26] outperformed Dejavu and the tool made by Juergens et.al with 14% to 17% better precision.

In contrast to current porting error detection analyses, our survey found some practical issues on porting activities in large scale project development. For example, one participant mentioned that original techniques for regression testing did not fit for backporting activities because of the large number of stable branches with different contexts. We believe that these issues from industry can greatly benefit further studies on clone detection and collaborative

development.

2.5 Linux Kernel Development Process

The community of Linux Kernel developers consists of approximately 5000 or 6000 members [35]. As of 2013, the 3.10 release of the Linux kernel had 15,803,499 lines of code. The main development branch in Linux Kernel (known as the mainline) is not a traditional"stable" branch. Instead, it incorporates all kinds of changes, both the latest features as well as security and bug fixes. The main branch is officially released as a new version approximately every three months, after several rounds of bug-fix pre-releases. At the beginning of each development cycle, which is around eight to twelve weeks, a two-week "*merge window*" is said to open and the changes will be merged into the mainline during this time, at a rate of approximate 1,000 changes per day. At the end of the "*merge window*", Linus Torvalds will declare that the window is closed and the first "-rc" kernels is released. Only bug-fixes will be accepted for the next six to ten weeks and Linus releases new "-rc" kernels around once a week till the series get up to between "-rc6" and "-rc9" and the kernel is considered sufficiently stable before the final "stable" release is made.

The mainline moves so fast that not all features are well tested before they get released [41]. For users who do not want to risk updating to new versions which may contain code that is not well tested, a separate set of stable branches exist, which are meant for people who simply want the security and bug fixes, but not a whole new version.

Once a "stable" release is made in the mainline, a corresponding stable branch is created under the *-stable* tree at `git://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/`. This stable branch will receive bug-fixes on "*as-needed*" basis for two to three months until the next mainline kernel becomes available [40]. Some versions of Linux Kernel are designated to be long term kernels which will receive support for a longer period. As mentioned in the development process guidance of Linux Kernel [41], "*the selection of a kernel for long-term support is purely a matter of a maintainer having the need and the time to maintain that release*". Generally, the maintainers of Linux Kernel select only one long term kernel per year "*as-needed*" [41].

The online guidance of stable branch development of Linux Kernel [42] describes how to submit backporting patches to stable branches. Developers are not allowed to add the patches they want to port to the *-stable* tree by themselves. Instead, they should include a tag—`stable@vger.kernel.org`, in the `sign-off` area. All required patches that are related to the selected patch should be included in the `sign-off` area, following the format below:

```
Cc: <stable@vger.kernel.org> # 3.3.x: a1f84a3: sched: Check for idle
```

```
Cc: <stable@vger.kernel.org> # 3.3.x
```

The tag sequence is equal to applying commands

```
git cherry-pick a1f84a3
```

```
git cherry-pick <this commit>
```

`Cherry-pick` is a `git` command that simply applies a patch from one branch to another. `Rebase` is another `git` command which supports porting changes between branches. Different from `cherry-pick`, it takes all the changes committed on one branch and replay them on another one [33]. These two commands are widely used for backporting.

The submitted patch is first posted to a relevant mailing list and reviewed by the developers in that list. After the patch is accepted by the corresponding subsystem maintainer who subscribes the mailing list, it will go into the *-next* tree of the subsystem. After the patch is merged to the *-next* tree of the subsystem, it receives more extensive reviews with integration tests. These *-next* tree of the subsystems will not be merged to the mainline until the "*merge window*" is open. After confirmed by subsystem maintainers, a successful patch are then merged into the mainline repository eventually.

After a new "stable" version of the mainline is released, the maintainer of *-stable* tree (currently Greg Kroah-Hartman) will find all patches with the tag `stable@vger.kernel.org` in the `sign-off` area, and all related patches mentioned in the `sign-off` area. He applies these patches to the stable branches and tests whether there are compilation errors after adopting the changes. If there is no compilation error, he adds these patches to the stable branches he maintains, and this process is a current recommended backporting process.

To further understand how developers actually backport patches, we designed a survey and asked developers how they decided which patches should be backported to stable branches and which information would be helpful to make this decision.

Chapter 3

A Version History Analysis of Backporting in Linux Kernel

To meet requirements from users who mainly concern the reliance rather than new features, Linux Kernel developers maintain multiple stable branches and frequently backport bug-fixes from the main development branch to stable branches. To understand the extent and manual effort involved in porting changes from the main development branch to maintenance branches, we studied 8 years (from 6/17/2005 to 10/31/2012) backporting activities in Linux Kernel. We analyzed the mainline and 35 stable branches from the version 2.6.12 to 3.6 during this time frame. Table 3.1 shows the details of the branches, active periods, and the total number of patches in our study.

Section 3.1 describes our approach to probe backporting events across the mainline and stable branches. We evaluate our approach in Section 3.2 with a ground truth dataset generated from the Linux bug repository *bugzilla*. Finally, we describe our study results in Section 3.3.

Branch	First Commit	Last Commit	Active Days	# of Patches
Mainline	2005-04-16	2012-10-31	2755	361311
Stable 2.6.12	2005-06-17	2005-08-29	73	54
Stable 2.6.13	2005-08-27	2005-12-15	109	47
Stable 2.6.14	2005-10-27	2006-01-30	95	97
Stable 2.6.15	2005-12-30	2006-03-27	84	116
Stable 2.6.16	2006-03-19	2008-07-21	854	1055
Stable 2.6.17	2006-06-11	2006-10-13	118	215
Stable 2.6.18	2006-09-19	2007-02-23	157	241
Stable 2.6.19	2006-11-29	2007-03-02	93	192
Stable 2.6.20	2007-02-03	2007-10-17	255	456
Stable 2.6.21	2007-04-23	2007-08-04	101	200
Stable 2.6.22	2007-07-08	2008-02-25	232	380
Stable 2.6.23	2007-10-09	2008-02-25	139	313
Stable 2.6.24	2008-01-23	2008-05-06	103	253
Stable 2.6.25	2008-04-17	2008-11-10	208	496
Stable 2.6.26	2008-07-10	2008-11-10	120	360
Stable 2.6.27	2008-10-09	2012-03-17	1255	1940
Stable 2.6.28	2008-12-23	2009-05-02	129	622
Stable 2.6.29	2009-03-22	2009-07-02	101	388
Stable 2.6.30	2009-06-09	2009-12-03	177	445
Stable 2.6.31	2009-09-05	2010-07-05	299	833
Stable 2.6.32	2009-12-02	2012-10-07	1040	3583
Stable 2.6.33	2010-02-24	2011-11-07	621	1880
Stable 2.6.34	2010-05-16	2012-08-20	827	1877
Stable 2.6.35	2010-07-30	2011-08-01	365	1615
Stable 2.6.36	2010-10-20	2011-02-17	120	685
Stable 2.6.37	2011-01-04	2011-03-27	82	594
Stable 2.6.38	2011-03-14	2011-06-03	80	667
Stable 2.6.39	2011-05-18	2011-08-03	76	448
Stable 3.0	2011-07-21	2012-10-31	468	2749
Stable 3.1	2011-10-19	2012-01-18	86	709
Stable 3.2	2012-01-04	2012-10-30	299	2381
Stable 3.3	2012-03-17	2012-06-01	74	700
Stable 3.4	2012-05-16	2012-10-31	164	1410
Stable 3.5	2012-07-21	2012-10-13	83	821
Stable 3.6	2012-09-28	2012-10-31	31	448

Table 3.1: Life span and the number of patches of each branch

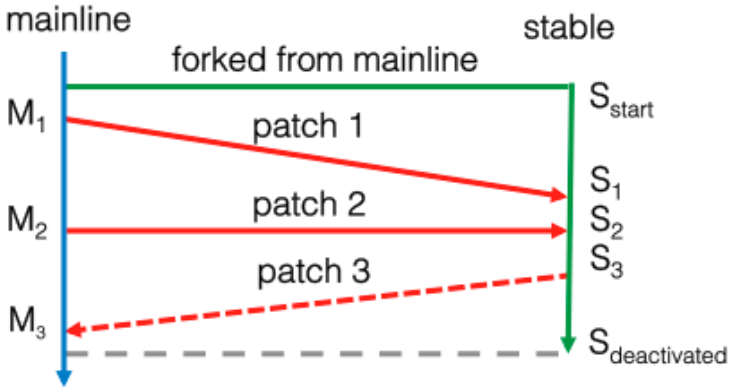


Figure 3.1: Backporting schematic diagram

The red arrows represent patch flows between the mainline and stable branches. Patch1 and Patch2 are considered as backported patches. Patch3 indicates upstream porting from stable branches to the mainline and is not considered in our backporting study.

3.1 Approach

Figure 3.1 illustrates the backporting activities in Linux Kernel. Once a version v is released in the mainline, a stable branch S is forked from the mainline branch according to the development process model described in Section 2.5. The stable branch S continues evolving from time S_{start} to time $S_{deactivated}$ when the branch S stops accepting changes.

We consider the time period from S_{start} to $S_{deactivated}$ as the *active period* of S. The vertical arrow lines indicate the timeline of the mainline and the stable branch S correspondingly. M_1, M_2, and M_3 represent the time when three mainline patches are created correspondingly, while S_1, S_2, and S_3 imply the time when three patches are created from the stable branch S. In

`git` patches, the time when the patch is generated is called the "author date", and the time when the patch is actually applied to the branch is called the "commit date".

We identifed patch propagation from the mainline to stable branches in the following three steps.

3.1.1 Step 1. Identify the Active Period of Each Stable Branch

We used a set of `git` commands to detect the active period of each stable branch. For a stable branch S, its start time S_{start} is identified using the `git` command: `git rev-list --merges --boundary --format="%cd" S`. This command lists all the merged commits in a reverse chronological order. We then grabbed the first merge commit as it indicated the forking point of the stable branch S. We marked its commit date as S_{start}. To identify $S_{deactivated}$, we used `git log -1 --format="%cd" S` to obtain all logs and then select the commit date of the last committed patch. Our results of active periods are consistent with the ones in *Wikipedia* as well [36]. The active period of each branch is listed in the Table 3.1.

3.1.2 Step 2. Collect Patches from the Mainline and Stable Branches within Active Periods

We retrieved patches from the mainline and the stable branch S that are committed within the active period of S.We excluded the commits of merge operations in `git` repository as these commits did not have any change difference (*diff* content) in the patch. We finally retrieved 29,270 patches from

35 stable branches and 361,311 patches from the mainline. The number of patches for each branch is also listed in Table 3.1.

3.1.3 Step 3. Identify Backporting Patch Pairs

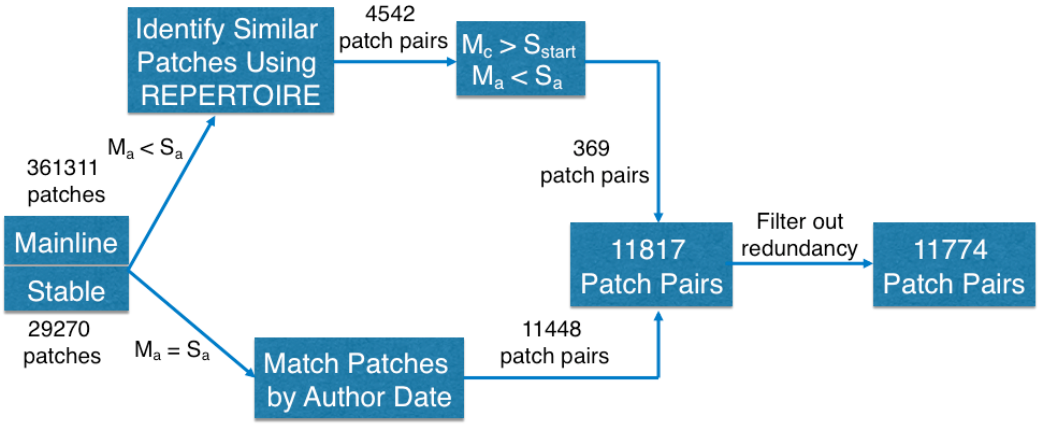


Figure 3.2: Data aquisition process for our empirical study

If a patch in stable branches has a similar change with a mainline patch [25], we consider these two patches as a backporting pair if the mainline patch appears before the other patch in stable branches or both patches appear at the same time ($M_a <= S_a$, where M_a and S_a represent the author dates of the patches in mainline and stable branches respectively). Patch1 and Patch2 represent backported patches in Figure 3.1. With the patches acquired in Step 2, we identified backporting patch pairs from the mainline and stable branches in the following two approaches summarized in Figure 3.2. With 29,270 patches from 35 stable branches and 361,311 patches from the mainline, we found a total of 11,774 backporting patch pairs.

(a) $\mathbf{M_a = S_a}$: Compare author dates of the patches from the mainline and stable branches. Based on the assumption that the patches generated by the same author at the same time are the same, we matched the patches in stable branches with the ones in the mainline by their author dates and authors (see Patch2 in Figure 3.1). According to the development process of Linux Kernel described in Section 2.5, we believed that most mainline patches in these patch pairs are ported to stable branches with `git` command `cherry-pick` and `rebase`. While `cherry-pick` applies a single patch from one branch to another, `rebase` takes all the changes that were committed on one branch and replay them on another one [34]. Backporting patch pairs identified in this approach are identical with their original ones in the mainline. This heuristic approach found 11,448 patch pairs in the case of $M_a = S_a$.

(b) $\mathbf{M_a < S_a}$: Use a clone detection tool to identify backporting events We made use of REPERTOIRE [25], a cross system porting analysis tool to help us search for the backporting patch pairs in which the mainline patch is created before the patch in stable branches $M_a < S_a$. In this case (see Patch1 in Figure 3.1), the stable patches are more likely to be edited manually before adapted to stable branches. The content, author, and author date of the patch from stable branches can all be different from the original mainline patch.

Considering that patches can only be backported to S from the mainline after S has been forked at time S_{start} (see Figure 3.1), we extracted the patches from stable branches that were generated within the active period of

S (i.e., $S_a > S_{start}$ and $S_a < S_{deactivated}$), then compared these patches with the mainline patches that appear in the same active period and prior to the stable patch (i.e., $M_a < S_a$ and $M_a > S_{start}$).

By setting a small similar token threshold, REPERTOIRE may over-estimate backporting changes though there is no semantic similarity in context between the two patches. When we select a higher token threshold, we may miss some small pieces of porting edits. Based on the accuracy evaluation of the prior work [25], we selected the token size 40 and find 369 backporting patch pairs in Linux Kernel when $M_a < S_a$.

REPERTOIRE regards similar code edits as cross-branch backported changes. However, similar code changes might not always be backporting events. There may exist multiple mainline patches that are found to be similar to another patch in stable branches, while only the latest one should be the porting source of the backported patch in stable branches. For instance, five mainline patches are detected to be similar with commit e3a5cb6 in Linux 2.6.32 by REPERTOIRE and all five patches meet the criteria mentioned above, yet only the last patch should be the reference implementation for the stable change and should be regarded as the propagation source of commit e3a5cb6.

We filtered out some false positive backporting patch pairs based on this analysis and identified 11774 backporting patches pairs with 11774 backported patches in stable branches.

3.2 Accuracy Evaluation

A segment of mainline patch	A segment of stable patch linux-2.6.32.y

Loc: drivers/staging/hv/channel.c
AuthorDate: Nov 8 14:04:38 2010
Author: Haiyang Zhang
Commit: Greg Kroah-Hartman
CommitDate: Nov 9 16:42:09 2010
Summary: staging: hv: Convert camel cased struct fields in channel_mgmt.h to lower cases
 	
`- if (channel->OfferMsg.)`
`MonitorAllocated {`
`+ if (channel->offermsg.`
`monitor_allocated) {`
`- set_bit(channel->OfferMsg.`
`ChildRelId & 31,`
`+ set_bit(channel->offermsg.`
`child_relid & 31,`
`(unsigned long *)`
	Loc: drivers/staging/hv/channel.c
	AuthorDate: Mar 21 14:41:37 2011
	Author: Olaf Hering
	Commit: Greg Kroah-Hartman
	CommitDate: Apr 14 16:53:23 2011
	staging: hv: use sync_bitops when interacting with the hypervisor. [Backported to 2.6.32 stable kernel by Haiyang Zhang]
	`if (channel->OfferMsg.`
	`MonitorAllocated) {`
	
	`- set_bit(Channel->OfferMsg.`
	`ChildRelId & 31,`
	`+ sync_set_bit(Channel->`
	`OfferMsg.ChildRelId & 31,`
	`(unsigned long *)`

Table 3.2: A porting example from the mainline to linux 2.6.32.y stable branch

To assess the accuracy of our approach, we manually constructed a ground truth of backporting patch pairs by checking the Linux Kernel bug repository *bugzilla*. We manually investigated 347 resolved bug reports in the mainline to detect backported patch pairs using the keywords of which module the bug locates, the author name or the resolved date from the bug description. Apart from reviewing the patch description, we manually checked the edited lines of the patch pairs and decided whether they were backporting patch pairs or not. If a patch pair had more than 10 lines similar in both

content and edit operations, we regarded this pair as a backporting pair. For example, there is a bug reported in the bug repository on 08/21/2010 which entitles "*Hyper-V (hv) staging drivers fail with 'scheduling while atomic' bug*", and resolved at 12/16/2010, we investigated all patches in the mainline with keywords "*staging*" and "*scheduling*" from 08/21/2010 to 12/16/2010, and found the patch shown in Table 3.2 on the left, which was written on 11/8/2010 by Haiyang Zhang and committed on 11/9/2010 by Greg Hartman. We then identified the backporting patch in stable branches with the keywords and found the patch shown in Table 3.2 on the right, which was generated by Olaf Hering on 3/21/2011, and committed on 4/14/2011 by Greg Hartman. In particular, the stable patch specified that this patch was *backported to 2.6.32 stable kernel by Haiyang Zhang*, who was the author of the mainline patch. We confirmed this backporting pair by investigating the edit lines on both syntax and edit operations.

Using this method, we generated a ground truth dataset with 135 backporting patch pairs. Based on this backporting patch pair dataset, we then measured the precision and recall of our patch pair identification method described in the previous section. Suppose G represents the ground truth and R is the result from REPERTOIRE. The precision and recall are defined as follows:

Precision. The percentage of porting patches found by the backporting identification approach that is also presented in the ground truth, i.e., $\frac{G \bigcap R}{R}$.

Recall. The percentage of ground truth found by the backporting identification approach that is also presented in the ground truth, i.e., $\frac{R \bigcap G}{G}$.

Our approach had the precision of 89.1% and recall of 75.2% in identifying backporting events from the mainline to stable branches in Linux Kernel.

3.3 Study Result

In order to understand the characteristics of backporting process in Linux Kernel, we analyzed backporting lag times from the mainline to stable branches (Section 3.3.1), the percentage of ported changes out of all changes (Section 3.3.2), the percentage of backported patches that are applied without any modifications (Section 3.3.3), and repetitive effort to port a mainline patch to multiple stable branches (Section 3.3.4).

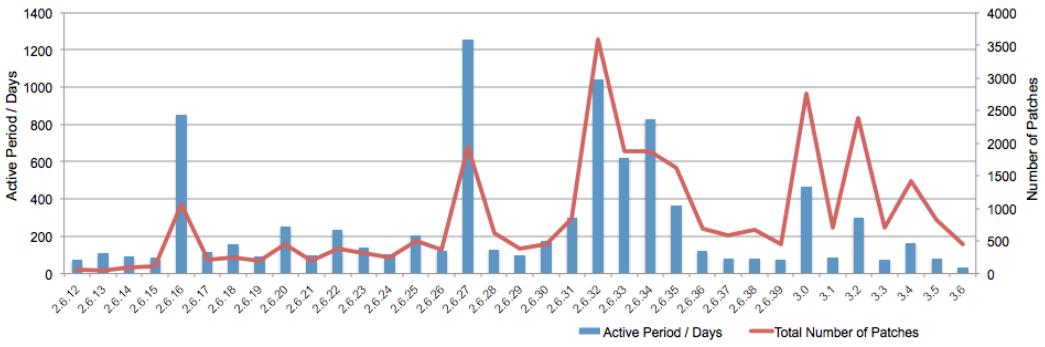


Figure 3.3: Life span and the number of patches of each branch

Figure 3.3 illustrates the life span for each branch based on Table 3.1. In this graph, the left Y-axis with the blue bars indicates the active periods of stable branches and the right Y-axis with red lines indicates the total number

of patches of each branch. The graph shows that long term branches tend to receive more changes.

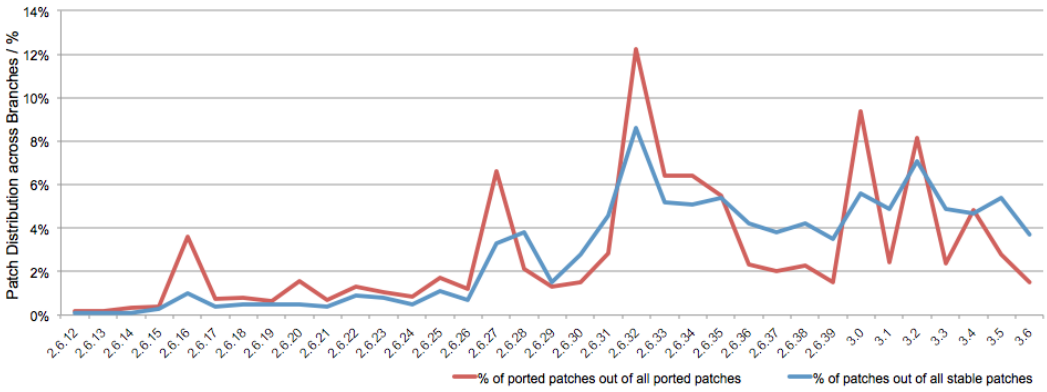


Figure 3.4: Patch distribution of stable branches

Figure 3.4 shows the patch distribution of all stable branches. The red line describes the backported patch distribution out of all backported patches, and the blue line graph shows the total patch distribution out of all patches. The graph illustrates a relatively consistent distribution of ported patches and all patches in each stable branch.

These two graphs mentioned above are further analyzed below.

3.3.1 How Long Does It Take for a Change to Get Propagated from the Mainline to Stable Branches?

It takes time to propagate patches from the mainline to stable branches. We defined porting time as the difference between the commit dates of the source patch in the mainline and the time the propagated patch is applied to stable branches. We then measured the average porting time for all propagated

patches for each branch. We also investigated the cumulative distribution of porting time for all ported patches in stable branches.

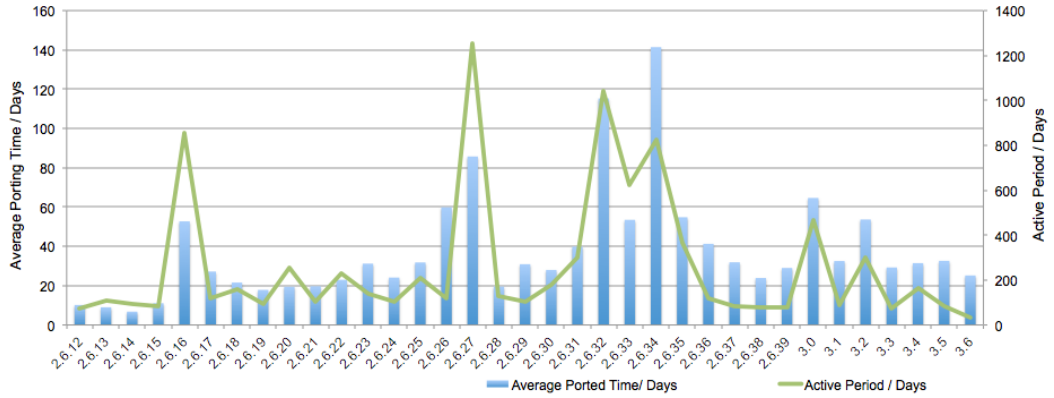


Figure 3.5: Porting time distribution of stable branches

Figure 3.5 describes the average porting time for all patches in each stable branch. The left Y-axis shows the average porting time and the right Y-axis represents the active period of each branch. The bar graph shows that the average porting time varies from 7 days to 141 days in 35 stable branches, with a median of 31 days and an average of 51 days. The green line graph describes the life span across all stable branches as a reference. The graph demonstrates that the average porting time is proportional to the active period of each branch in general.

Figure 3.6 shows the cumulative distribution of propagation time. The X-axis represents the porting time while the Y-axis illustrates the cumulative ratio of backported patches in all stable branches. In 11,774 patch pairs we identified, 60% of backported patches got distributed in 30 days, while at the end of 2 months, 80% of backported patches got propagated. We also noticed

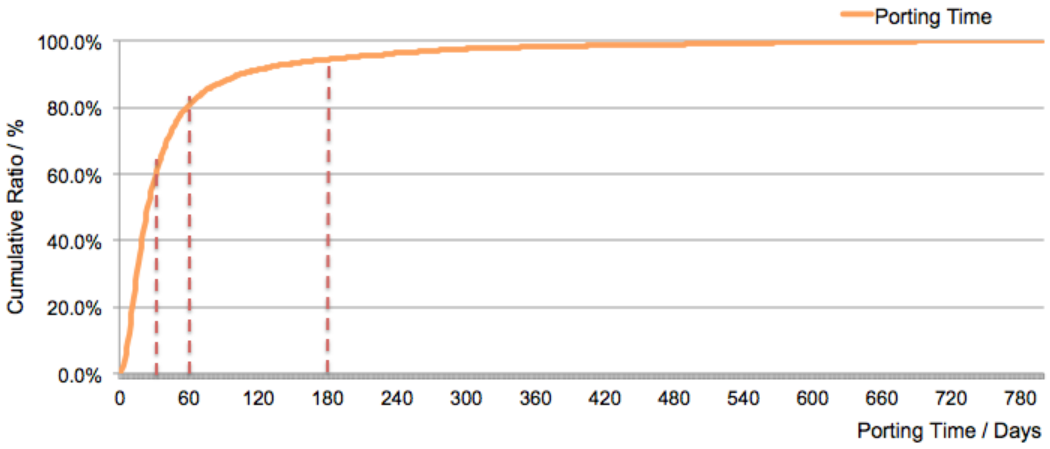


Figure 3.6: Cumulative distribution of porting time

that the maximum propagation time of porting events was more than 2 years and 2% patches were ported after 1 year. This result indicates that though most changes can be propagated quickly, some changes that need to ported are not propagated till very late.

3.3.2 What Percentage of Patches in Stable Branches Are Coming from the Mainline?

After investigating the propagation time of the backporting practice, we looked into the extent of ported changes out of all edits in stable branches.

Figure 3.7 displays the distribution of backported patches in 35 stable branches. The left Y-axis shows the number of patches, the blue bars indicate the number of propagated patches and the red bars illustrate the total number of patches in each stable branch. The right Y-axis with the green line graph shows the average backporting rate across branches. We defined the average

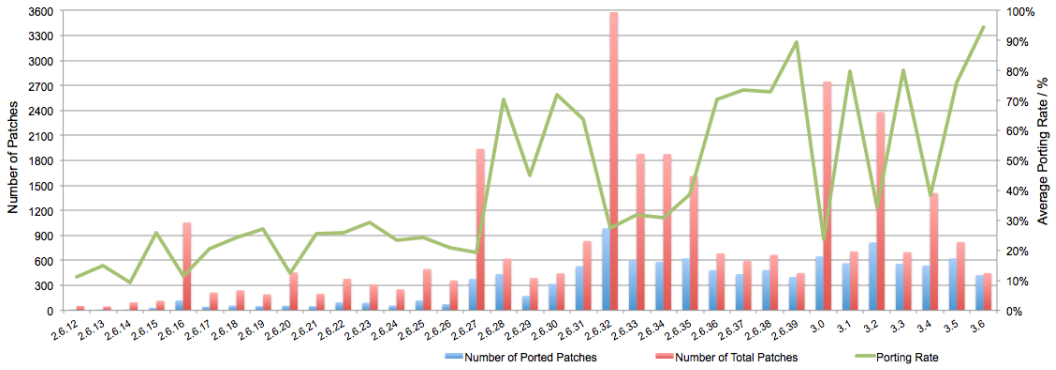


Figure 3.7: Porting rate distribution of stable branches

porting rate as the percentage of clone patches out of all patches in a stable branch. We found that the porting rate varied from 10% to 90%, with a median of 29% and an average of 41%, which indicated that a significant portion of changes in stable branches originate from the mainline.

We also noticed that, since the stable branch 2.6.27, the number of backporting patches for each branch remained relatively stable, while the total number of patches for each branch changed from time to time across branches. Thus the more changes a branch received, the lower its backporting rate was.

3.3.3 What Percentage of Backporting Patches Are Applied Without Any Code Adaptation?

When developers port changes from the mainline to stable branches, they must make sure that the change can be transformed consistently from the source to the target. We measured the adaptation degree during the process of change propagation by assessing the similarity between a reference patch

and a target patch.

We defined change similarity as the percentage of backported lines out of all revised lines in one patch, i.e.

$$\text{Change Similarity} = \frac{\sum portedLines}{\sum editedLines} \times 100\%$$

If the ported patch is exactly the same as the source patch, the similarity should be 100%. However, when the patch is revised to fit to the target branch, the change similarity is less than 100%.

We found that out of all backported patches in stable branches, 97.5% were propagated to stable branches without any modifications. The rest of backported patches which were manually modified also showed a change similarity of 60% on average. This result indicated that a large number of patches were applied to the target with little modification.

3.3.4 How Much Repetitive Effort Involve in Porting a Mainline Patch to Multiple Stable Branches?

A mainline patch might be ported to more than one branch, and in our study, we investigated how many branches a mainline change got propagated to and how many mainline patches were ported to more than one stable branch.

As shown in Figure 3.8, the Y-axis illustrates the number of patches while the X-axis is the number of branches the mainline patches get propagated to. The result showed that mainline patches were ported to 2.25 branches on average. In an extreme case, the changes were ported to 14 stable branches at most. Our result indicated that there was a significant repetitive effort on

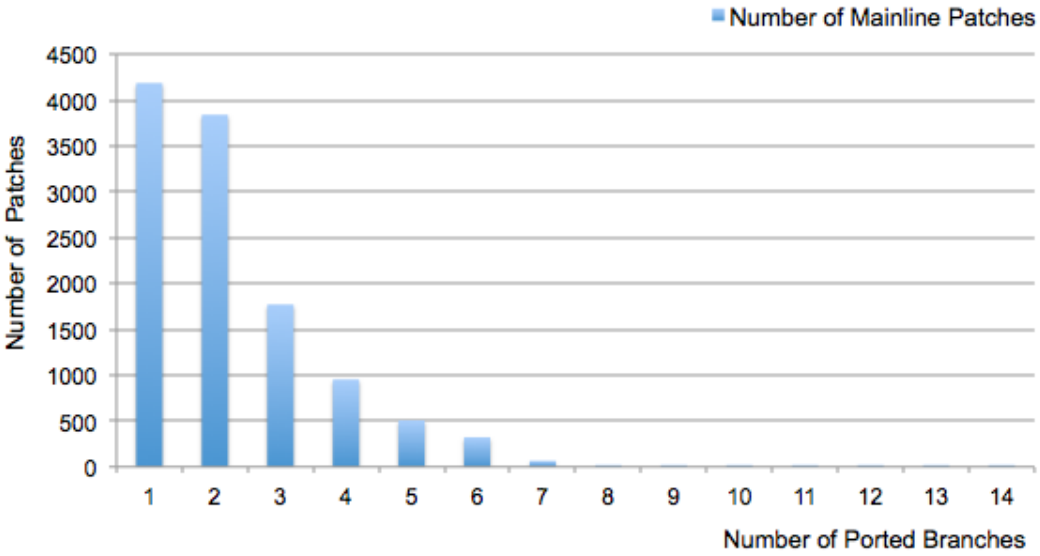


Figure 3.8: Distribution of backporting repetitive effort

cross-branch porting from the mainline to stable branches in Linux Kernel.

3.4 Summary

This section presents a version history study on cross-branch backporting from the mainline to stable branches based on the source code repository of Linux Kernel.

Our study found that a number of changes in the mainline got propagated to more than one maintenance branch repetitively with very little modification to fit for the target context. However, the maximum lag time for porting events was more than 2 years, indicating that some patches required non-trivial effort to propagate to the target branches.

The results we obtain call for an automated approach of applying similar program transformation to forked branches and notify developers of potential porting events.

Chapter 4

A Survey on Cross-Branch Backporting Practice in Linux Kernel

While our version history study quantified the extent and frequency of the cross-branch backporting practice in Linux Kernel, the analysis did not present how developers perceived the challenges of backporting and how they actually performed backporting across branches.

We carried out a follow-up survey to understand how developers determined a patch that should be ported, and how they modified the patches and apply them to the target branches. We purposely targeted 228 Linux Kernel engineers who might have ported code from the mainline to stable branches based on our analysis of the Linux Kernel repository described in Chapter 3. We sent an online survey entitled "*A Survey on Cross-Branch Change Awareness in Linux Kernel*" at `https://www.surveymonkey.com/s/seal_awarenessSurvey` to these developers. We received 22 responses and 14 of them completed all the questions.

According to the responses, our participants had over 20 years software development experience and 12.6 years experience on average in Linux development with C. As shown in Figure 4.1, 50% of them worked on the storage

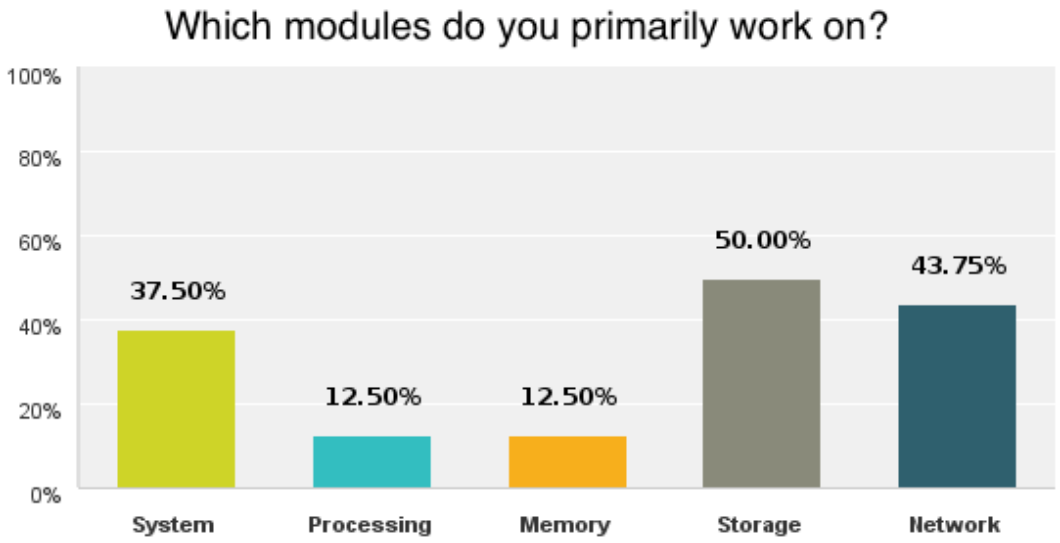


Figure 4.1: Modules that participants work on

module, 43.75% on the network, 37.5% of them on the system drivers, and the rest focused on processing and memory management.

The survey consisted of 15 questions. Section 4.1 describes developers' perception on the risks and effort of backporting practice. Section 4.2 describes how developers collect information, identify the patches that need to be propagated to other branches, and finally modify the selected patches before applying them to the target branches. We summarize our survey results and future work in Section 4.4.

4.1 What Do You Think of Cross-Branch Porting?

To understand how developer perceive the cross-branch porting, we listed eight statements about the changes and risks about backporting and

asked participants to rate their agreement on these statements.

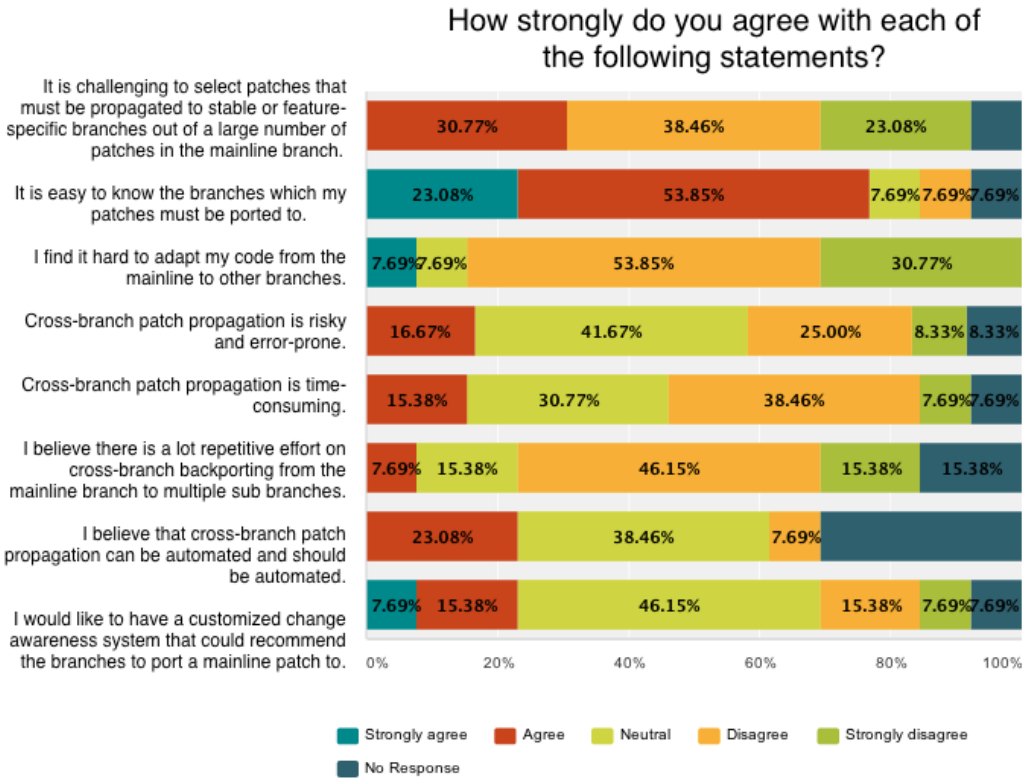


Figure 4.2: Developers' perception on the backporting activities

In Figure 4.2, the statements are listed on the left, while the percentage distribution for each opinion is shown as a stack bar graph on the right. From left to right, the parts of the strips in blue, red, yellow, orange, and green indicate the percentage of participants who strongly agree, agree, neutral, disagree, and strongly disagree with the statement on the left correspondingly.

We found that developers held diverging opinions with respect to the risks and difficulties of cross-branch porting. For example, one third of them

believed that it was hard to identify backporting patches, one third disagreed with this statement and the rest held a neutral stance on whether it was hard to perform cross-branch porting from the mainline to stable branches. 17% participants believed that backporting was risky and error-prone, 42% adopted a neutral position, and 33% disagreed with the statement.

When we asked "`what are the risks or challenges for backporting changes from the mainline to other branches?`", participants reported regression bugs, constraints omission and risks of introducing redundant dependency. The following quotes describe the challenges of the cross-branch porting practice:

"*You might miss an important detail and introduce a bug. And it (cross-branch porting) requires deep understanding of both the code being changed and the purpose and function of the patch.*"

Another participant mentioned that current techniques for regression testing failed to support the testing of backporting activities due to a large number of different branches.

"*Subtle changes in the surrounding code or the used APIs might invalidate the assumptions that the original change depended on, without any compiler error or warning. Ideally, the same tests originally used to validate the change should be re-run after it is applied to each of the other branches, but the large number of stable branches can make this impractical.*"

In summary, from developers' perspectives, most backporting can be

done with minor effort, yet some backporting practices are risky due to a lack of full understanding of the program. These faulty transformations may break the program dependency and introduce new bugs to the target branches.

4.2 How Do You Identify the Patches That Should Be Ported?

To address the risks and challenges of cross-branch porting, we investigated the process of backporting. We asked questions on how they collected necessary information about the backporting process, how they identified the patches that should be ported, and how they modified the patches and applied them to the target branches.

(1) How do you collect information about the development activities in the mainline?

As a first step to understand how developers identify patches that should be ported, we asked them how they collected relevant changes from the mainline and how they filtered out the irrelevant edits out of all change events.

As shown in Figure 4.3, we list five methods and we ask developers to rate how often they use these methods to be aware of the latest change activities in the mainline. The stack bar graph illustrates the distribution of the usage frequency with respect to the methods listed on the left. From left to right, the parts of strips in red, yellow, blue, and orange correspond to the percentage of participants who always, often, sometimes, and never use the

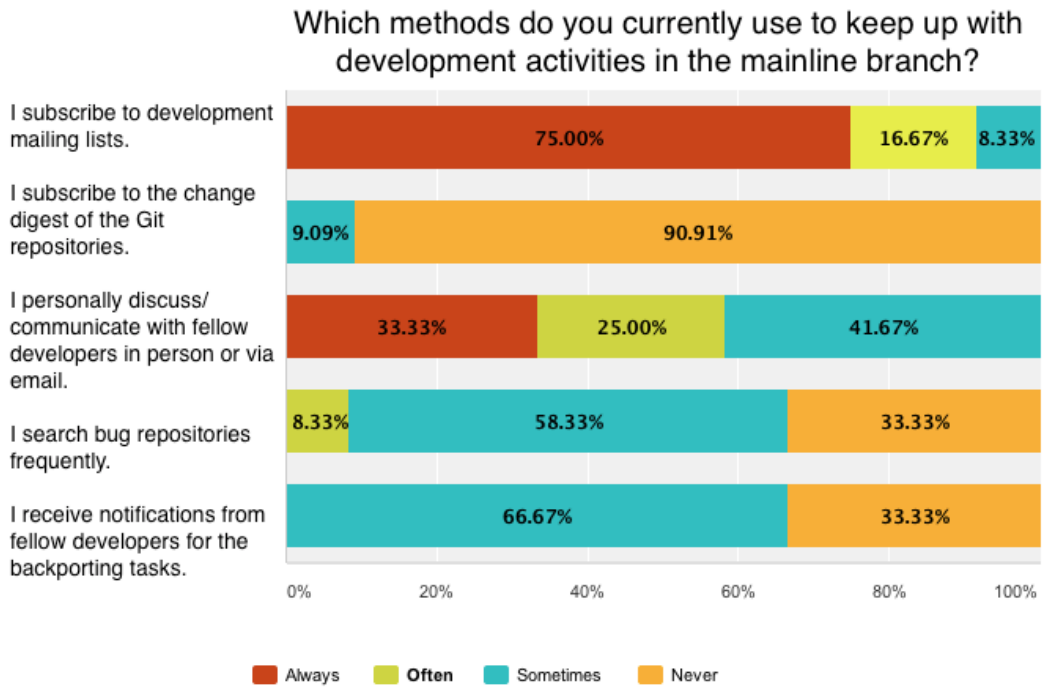


Figure 4.3: Methods that are used to know about development activities in the mainline

method mentioned on the left to know about the development activities in the mainline.

According to the graph, subscribing to a mailing list was the most important method to keep track of the latest development activities in the mainline. Developers also discussed with peer engineers and search for the bug repository to be aware of the activities. The quotes from participants reported that `LWN.net` [38], and the Linux Kernel news websites [44, 39, **?**] were also popular in the Linux Kernel development community.

To investigate which information may be useful to filter out irrelevant

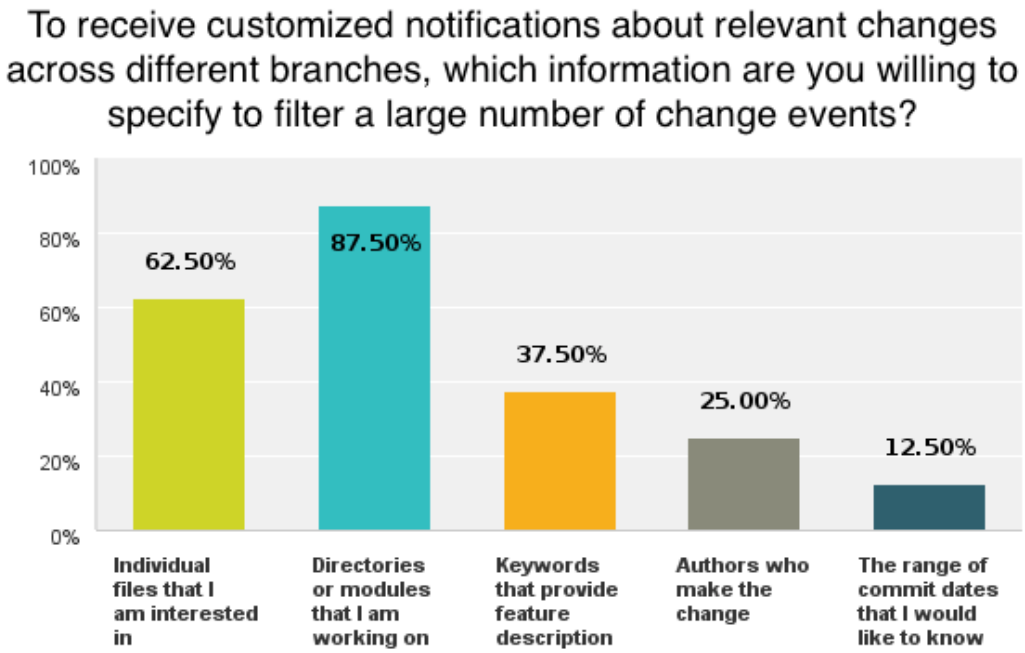


Figure 4.4: Information that helps developers filter out irrelevant change events

edits out of numerous edits in the mainline, we asked participants to select the information they prefer to specify for the customized notification of the relevant changes. Figure 4.4 illustrates that 87.5% of participants concerned about directory names, and another 62.5% chose to use the files that they were working with to customize change event notification. The range of commit dates and the authors who made the change might not be effective enough to filter out unrelated events.

(2) Which information can be useful to identify the patches that need to be ported?

After understanding how developers collected information about the

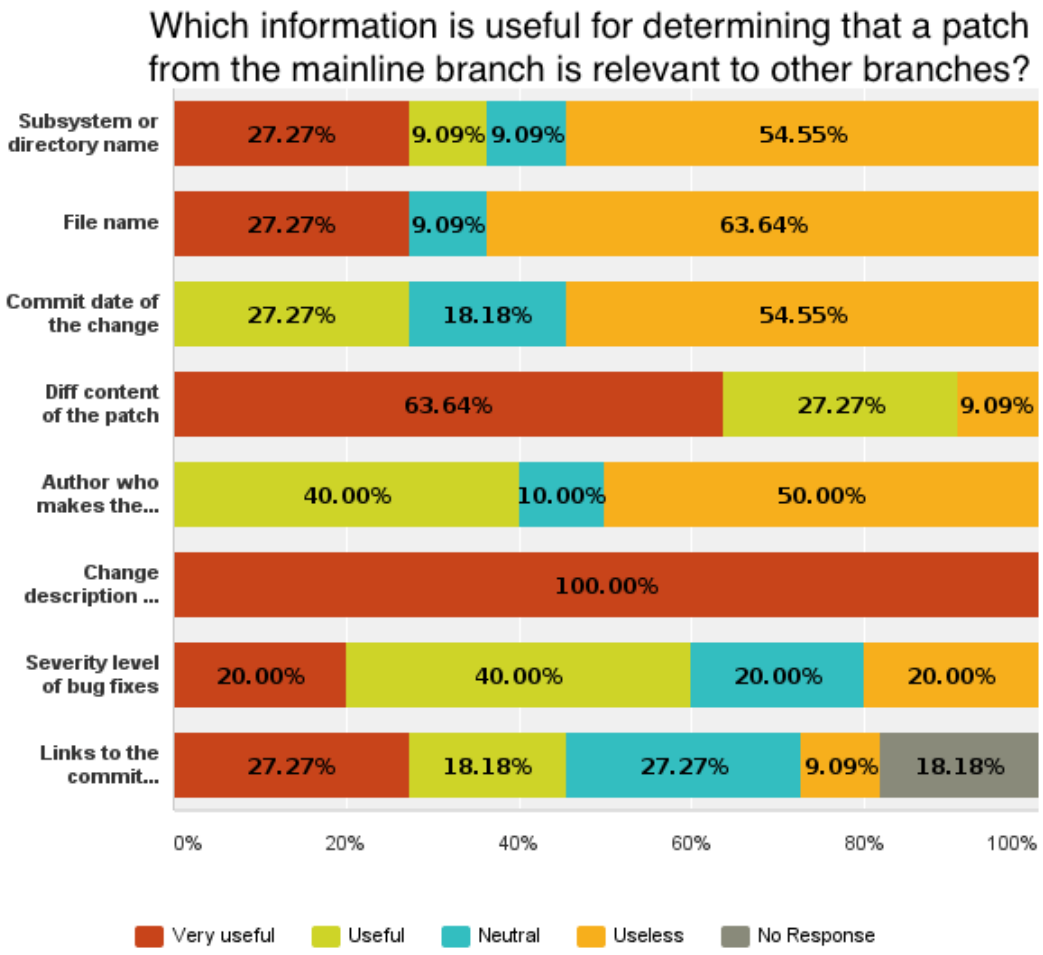


Figure 4.5: Useful information to identify the patches that should be ported

relevant changes, we investigated how they digested the information about relevant changes and decided whether to port the patches or not. We gave participants eight options shown in Figure 4.5 and asked them to select the kind of information they regarded as useful. The stack bar graph presents the distribution about the different preferences about the kind of information

they found useful for determining a patch to be ported. From left to right, the parts of the strips in red, yellow, blue, and orange indicate the percentage of participants who regard the information as very useful, useful, neutral, and useless correspondingly.

Based on the responses, 100% of participants believed that the change description is particularly useful in determining that a patch should be ported to stable branches. The *diff* content of the patch was another important information to identify the patches. However, the name of the changed file, the author of the patch, and the commit date might not be very critical for the decision of backporting a patch.

(3) How do you determine that a patch should be ported?

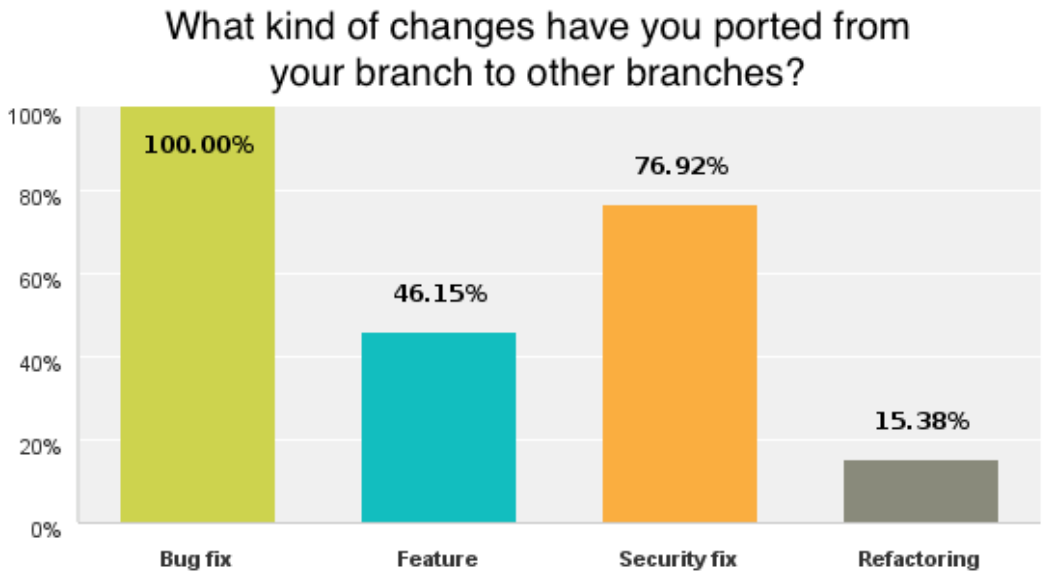


Figure 4.6: Types of ported changes

From the development process of Linux Kernel described in Section 2.5, we hypothesized that most ported changes are serious bug fixes. This assumption was proved by Figure 4.6. As shown in the graph, all participants reported that they ported bug fixes across branches. We also noticed that more than half of participants had ported features and refactorings to stable branches, indicating that a small potion of features and refactorings were also propagated along with serious bug fixes.

Regarding the last question to investigate how to identify the patches that need to be ported, we asked developers to provide insights on how they made this backporting decision. We found that apart from considering the severity of the bug fixes, developers also looked for the bugs introduced by their own commits and took the responsibility to port the bug fixes to maintenance branches. Some quotes are as below:

"*If the bug is serious (data loss, security vulnerability, crash, important functional regression), it should be ported.* "

"*I check when the bug I'm fixing was introduced, then ask for the fix to be applied to all active stable branches since that version.* "

In summary, developers subscribed to the mailing list or discussed with other developers to keep up with the development activities in the mainline. By investigating the change descriptions and *diff* content of the patch, developers judged the edit and relevant changes based on their experience, and decided whether they needed to backport this change to the target branches along with

relevant features and refactorings.

4.3 How Do You Modify the Patches to Fit for the Target Branches?

Patches need to be properly backported to the target branch. Regarding the last step of the backporting process, we investigated the types of modifications and the time effort to adapt a patch to individual target branches.

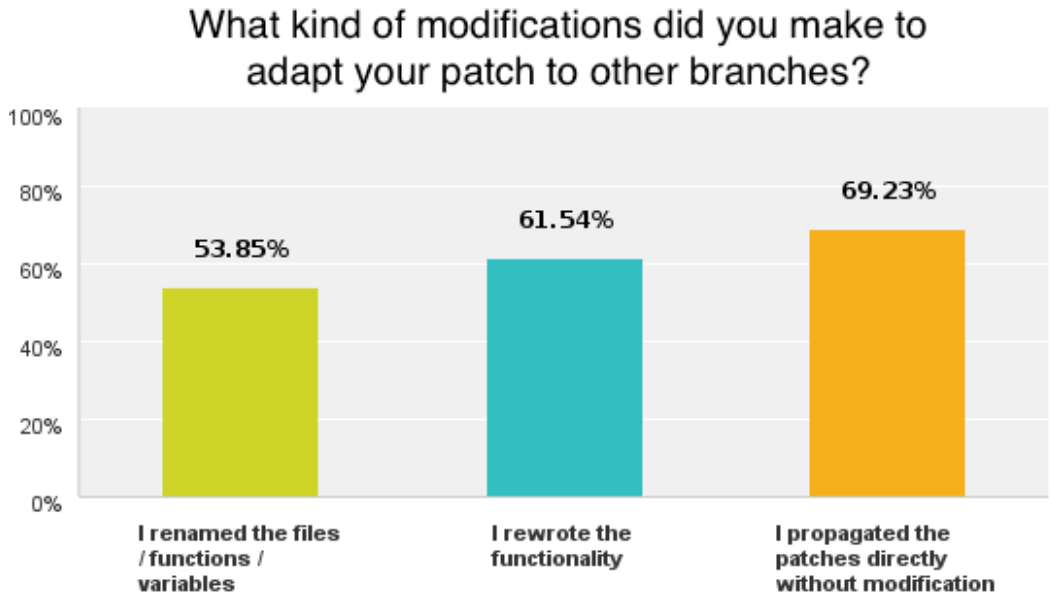


Figure 4.7: Adaptation types that are applied to backported patches

Based on the development process of Linux Kernel described in Section 2.5, we hypothesized that developers often apply the `git` commands such as `git-cherry-pick` and `git-rebase` to backport identical patches from the

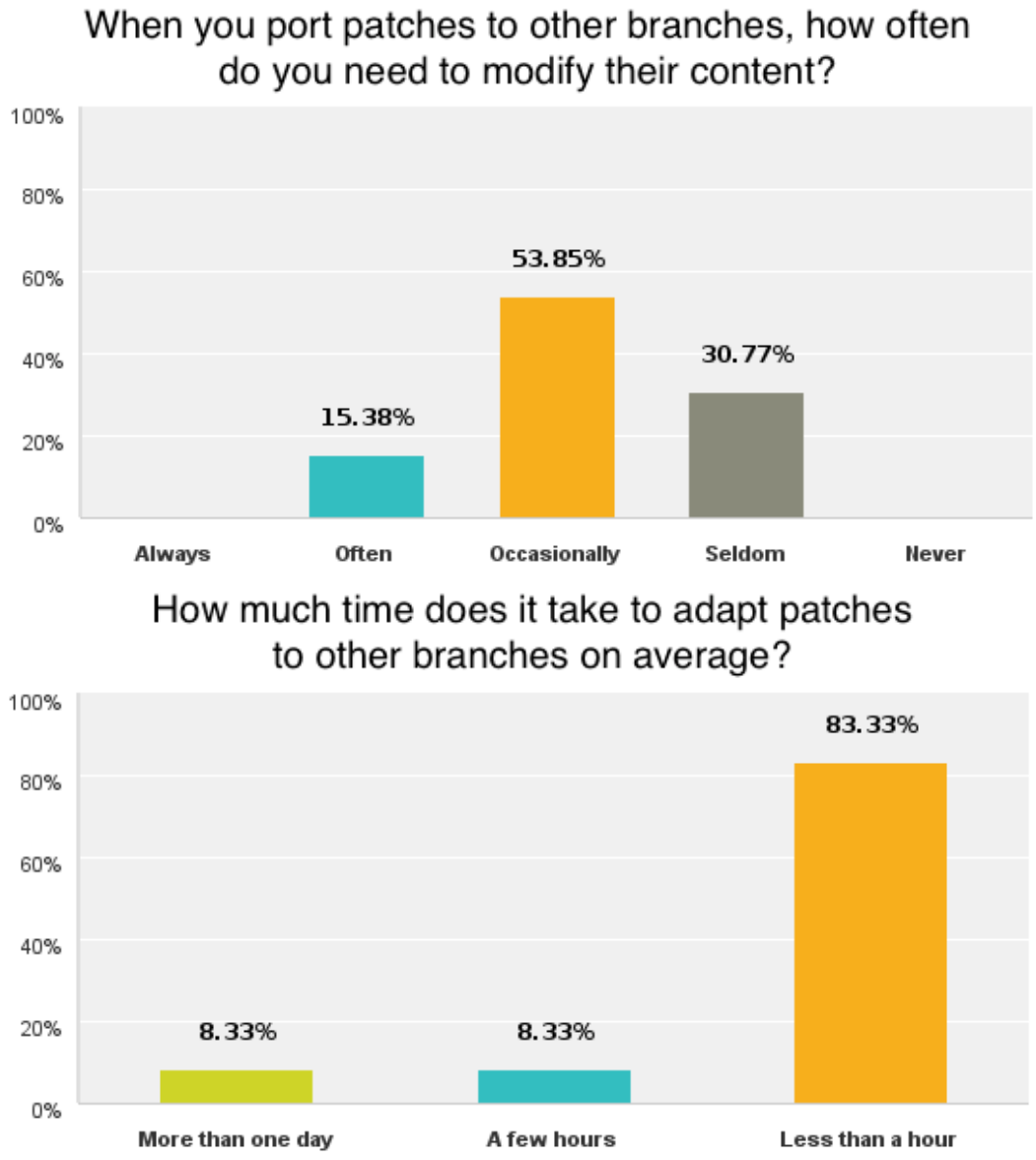


Figure 4.8: Time effort to adapt a patch across branches

mainline and replayed them on stable branches. Figure 4.7 illustrates the types of modification during the process of cross-branch porting. Aligned with our

hypothesis, around 70% of participants directly propagatee changes without any modification. For the rest of patches that needed to be occasionally modified to fit for the target branches, the modifications were simply renaming and function rewriting that could be done within approximately one hour (see Figure 4.8).

Lastly, for the developers who regarded it difficult to modify the patches and apply them on the target branches, we asked them to provide more insights on why it was hard to adapt a change to individual target branches. Participants reported that though most backporting jobs could be done with trivial effort, it was rather hard to make sure that every transformation was done consistently across branches without breaking existing constraints or introducing redundant dependency. This consistency was particularly difficult when restructuring and refactoring were involved in the backporting. The following quotes describe the challenges of adapting changes across branches.

"*Most cases are relatively obvious. Worst case is when changes in different parts of the kernel cause a run-time failure while the patch backporting itself required no specific effort.*"

"*backporting fixes / features across major refactoring / restructuring can be messy and you gotta judge whether and how much of such intervening restructuring / refactoring to slurp in together.*"

4.4 Summary

Our survey with the Linux Kernel developers presented that the risks and challenges were acknowledged by most participants of our survey, such as introducing redundant code or breaking existing constraints in the target branches. Most backporting could be done with minor manual effort. Considering that all of them had more than 10 years of experience on Linux Kernel development, we argued that the majority of novices or developers in the Linux Kernel development community were seldom given the responsibility to port a change from the mainline to stable branches. Due to the risks of missing relevant changes and breaking dependencies, cross-branch backporting was always done by the trusted and experienced experts such as subsystem maintainers and the owners of the operating system, who were aware of the change impact of the selected patch and able to identify all relevant changes that must be ported along with the selected one.

Our results call for an automate tool support to sift relevant change events out of a large number of change events, collect enough context about the patches, and notify pertinent developers who are responsible for the corresponding code propagation.

Chapter 5

Conclusions

This chapter summarizes our contributions, threats to validity, and future research directions.

In large scale projects, a main development branch moves so fast such that not all cutting edge features in the master branch are well tested. The users and vendors, who mainly focus on the stability and reliance rather than new features, are often skeptical about using the less stable mainline branch. A common practice to solve this problem is to maintain a main development branch and multiple maintenance purpose branches that accept serious bug-fixes simultaneously. Bug-fixes are frequently applied from the development branch to the maintenance branches and this process is known as *backporting*.

We performed two complementary studies on the backporting practice in Linux Kernel. As the first step, we conducted a study investigating 8 years of version history data from the mainline and 35 stable branches in Linux Kernel. We identified 11,774 propagated patch pairs from the mainline to stable branches with the precision of 89.1% and recall of 75.2%. We then analyzed the extent and propagation effort of the backporting activities and found that 60% of the patches in stable branches were propagated from the

mainline, with an average porting time of 51 days. Most propagated changes were ported with minor modifications to more than two stable branches on average. However, there also existed around 2% of backported changes which took more than 1 year to propagate to the target branches.

To further reason about how developers perceive the challenges and risks of backporting and how much effort they exert for the backporting practice, we carried out a follow-up online survey with the developers who may have ported code based on our version history analysis. We received 22 responses from Linux Kernel experts who had more than 10 years experience in Linux Kernel development on average. Our results indicated that developers often subscribed to the mailing lists and communicated with peer programmers to identify the patches that should be backported. We also found that inexperienced developers seldom had an opportunity to port changes from the mainline to stable branches due to a lack of global understanding of the impact of the patch to be ported. While most changes could be performed safely with little effort, some faulty transformation might easily break code dependencies and introduce new bugs to the target branches.

Based on the results above, we argue that the majority of developers who do not have enough knowledge to identify the patches that need to be ported along with all relevant changes, have a need for automated tool support to identify the patches that should be ported and to become aware of the relevant changes that need to apply together with the selected patches.

5.1 Threats to Validity

For our version history analysis, we used the clone detection tool REPERTOIRE to help us locate relevant patch pairs. REPERTOIRE uses the widely used clone detector CCFinderX [14] to identify similar code edits. By setting a lower token threshold of CCFinderX, the tool REPERTOIRE may over-approximate potential backporting changes. When we select a higher threshold, we may miss some small pieces of porting edits. Ray et al. [25] conducted an accuracy evaluation experiment in the BSD family, indicating that 40 tokens could reach the best F-measure result for both precision and recall. Considering that the members of the BSD family are also large-scale operating systems implemented in C, which is similar to Linux Kernel, we selected the token size 40 in our version history study.

In our version history study, we concentrated on cross-branch porting effort from the main development branch to multiple maintenance branches in Linux Kernel. We acknowledge that our version history study on the mainline and stable branches may not generalize to other cross-branch porting, such as porting from feature-specific branches to the main branch or upstream porting from stable branches back to the mainline. Yet the backporting pattern in Linux Kernel is likely to be found on other open source projects. This pattern will be valuable for other development process models such as co-evolving product variants in software product lines.

Regarding our survey on the developer perceptions of backporting, we notice that all of our participants have more than 10 years experience in Linux

Kernel and most of them are subsystem maintainers in either Linux Kernel or related projects such as SUSE and Red Hat. They are not representative of the majority of Linux Kernel developers who only contribute to Linux Kernel occasionally. The fact that we only received responses from experienced developers indicates that the inexperienced developers, who are the majority of the Linux Kernel development community, seldom perform backporting due to the risks and difficulties involved, thus the backporting practice is confined to the experts who have a relatively thorough understanding of the program. This bias driven by participants' experience level may impact our results about the effort of backporting.

Maintenance branches in large scale systems may have some unique characteristics that are different from other feature-specific branches. According to the development process of Linux Kernel, only serious bug-fixes that have already been merged to the mainline are included in stable branches. Another special characteristic of the maintenance branches is their relatively short active periods. Except four to five long term stable branches, the life spans of stable branches in Linux Kernel are as short as two to three months on average. Due to the short evolution period, most stable branches are quite similar to the mainline and do not have their own particular features but propagated patches. While we acknowledge that our study on the mainline and stable branches in Linux Kernel may not generalize to other branches in different systems, we argue that our results on the cross-branch backporting are meaningful considering that Linux Kernel is a typical long-surviving large

scale system and its development process model is adopted by numerous large scale systems.

5.2 Future Work

Leveraging the study results above, a change awareness and notification system for cross-branch backporting can be built to locate changes that need to be propagated and notify relevant developers with appropriate recommendations about relevant changes that should be ported together.

Second, since our study only focuses on the cross-branch backporting from the mainline to stable branches, some other empirical studies should be done further to analyze code porting activities from feature-specific branches to the main development branch or upstream porting from stable branches to the mainline.

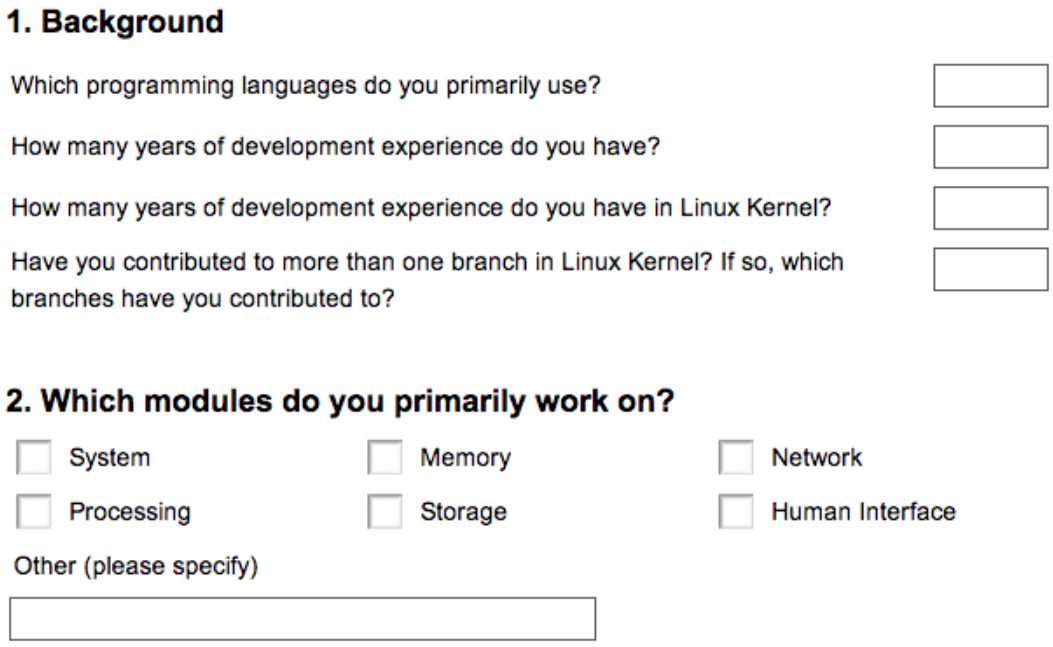
Lastly, the development of open source products currently lacks strict specifications, urgent bug-fix tasks, and formal development process. It would be interesting if more cross-branch porting studies can be done in industrial projects where a strict guideline for the backporting practice exists.

Appendix

Appendix 1

Survey Questions

In this appendix, we show the screenshot for the online survey.



1. Background

Which programming languages do you primarily use?

How many years of development experience do you have?

How many years of development experience do you have in Linux Kernel?

Have you contributed to more than one branch in Linux Kernel? If so, which branches have you contributed to?

2. Which modules do you primarily work on?

System | Memory | Network

Processing | Storage | Human Interface

Other (please specify)

3. How strongly do you agree with each of the following statements?

	Strongly agree	Agree	Neutral	Disagree	Strongly disagree	No Response
It is challenging to select patches that must be propagated to stable or feature-specific branches out of a large number of patches in the mainline branch.	○	○	○	○	○	○
It is easy to know the branches which my patches must be ported to.	○	○	○	○	○	○
I find it hard to adapt my code from the mainline to other branches.	○	○	○	○	○	○
Cross-branch patch propagation is risky and error-prone.	○	○	○	○	○	○
Cross-branch patch propagation is time-consuming.	○	○	○	○	○	○
I believe there is a lot repetitive effort on cross-branch backporting from the mainline branch to multiple sub branches.	○	○	○	○	○	○
I believe that cross-branch patch propagation can be automated and should be automated.	○	○	○	○	○	○
I would like to have a customized change awareness system that could recommend the branches to port a mainline patch to.	○	○	○	○	○	○

4. Have you ever backported a bug fix or feature from the mainline branch and applied it to stable or feature-specific branches? If so,

How do you determine that your patch should be ported to other branches? Please provide rationales based on your experience.	
How do you identify the branches your patch should be backported to?	
How many hours per month do you spend learning about the development activities in branches other than the branch you generally work on?	
How many hours per month do you actually spend backporting patches from your branch to other branches?	

5. What kind of changes have you ported from your branch to other branches?

☐ Bug fix ☐ Feature ☐ Security fix ☐ Refactoring

Other (please specify)

6. When you create a patch, and determine that a patch must be ported to other branches, what do you do? Please select all that apply.

- [ ] I generally take the responsibility of cross-branch porting by myself.
- [ ] I notify the developers who are responsible for the target branches.
- [ ] I just specify in the commit that this patch should be or might be ported to other branches.
- [ ] I do nothing for the cross-branch porting.

Other (please specify)

7. When you port patches to other branches, how often do you need to modify their content?

- ○ Always
- ○ Often
- ○ Occasionally
- ○ Seldom
- ○ Never

8. How much time does it take to adapt patches to other branches on average?

- ○ More than one day
- ○ A few hours
- ○ Less than a hour
- ○ Not applicable (I never modify the porting patches or do not know).

9. If you have ever modified patches to fit the content of other branches,

What percentage of cross-branch porting can you apply with minimal effort (such as a direct application of git-cherry-pick or git-rebase)?

If you have found the process of patch content modification difficult, could you please provide insights on why?

10. What kind of modifications did you make to adapt your patch to other branches?

- [ ] I renamed the files / functions / variables
- [ ] I rewrote the functionality
- [ ] I propagated the patches directly without modification

Other (please specify)

11. What risks/challenges do you think are associated with backporting changes from the mainline branch to other branches?

12. Which methods do you currently use to keep up with development activities in the mainline branch?

	Always	Often	Sometimes	Never
I subscribe to development mailing lists.	○	○	○	○
I subscribe to the change digest of the Git repositories.	○	○	○	○
I personally discuss/communicate with fellow developers in person or via email.	○	○	○	○
I search bug repositories frequently.	○	○	○	○
I receive notifications from fellow developers for the backporting tasks.	○	○	○	○

Please list all other tools or approaches you have used or think will be useful to know about development activities in the mainline branch.

13. Which information is useful for determining that a patch from the mainline branch is relevant to other branches?

	Very useful	Useful	Neutral	Useless	No Response
Subsystem or directory name	○	○	○	○	○
File name	○	○	○	○	○
Commit date of the change	○	○	○	○	○
Diff content of the patch	○	○	○	○	○
Author who makes the change	○	○	○	○	○
Change description or commit message	○	○	○	○	○
Severity level of bug fixes	○	○	○	○	○
Links to the commit description in Git repositories	○	○	○	○	○

Please let us know any other information about the Linux Kernel which you think is important to determine that a patch from the mainline branch is relevant to other branches.

14. Which information delivery mechanisms do you prefer to keep up to date with relevant changes across branches?

	Strongly preferred	Preferred	Neutral	Not Preferred	No Response
Email subscription	○	○	○	○	○
Website that you can log-in and get personalized news feed (like Facebook)	○	○	○	○	○
RSS Feed	○	○	○	○	○
Public website that you can search and browse	○	○	○	○	○

Please list other methods you may like to learn about the development activities in the mainline branch.

15. To receive customized notifications about relevant changes across different branches, which information are you willing to specify to filter a large number of change events? Check all your preference.

- ☐ Individual files that I am interested in
- ☐ Directories or modules that I am working on
- ☐ Keywords that provide feature description
- ☐ Authors who make the change
- ☐ The range of commit dates that I would like to know

Other (please specify)

Bibliography

[1] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proceeding of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2012*, volume 45, 2012.

[2] Christian Bird, Thomas Zimmermann, and Alex Teterev. A theory of branches as goals and virtual teams. In *Proceeding of the 6th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2012*, pages 301–310, 2012.

[3] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A theory of software product line refinement. In *Proceeding of the European Conference on Software Maintenance and Reengineering, CSMR 2013*, pages 25–34, 2013.

[4] Hongyu Pei Breivold, Stig Larsson, and Rikard Land. Migrating industrial systems towards software product lines: Experiences and observations through case studies. In *Proceeding of the 16th Fundamental Approaches to Software Engineering, FASE 2013*, pages 285–300, 2013.

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceeding of the 19th ACM*

SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2011, pages 168–178, 2011.

[6] J.R. Cordy. Exploring large-scale system similarity using incremental clone detection and live scatterplots. In *Proceeding of the 19th IEEE International Conference on Program Comprehension, ICPC 2011*, pages 151–160, 2011.

[7] Yael Dubinsky, Julia Rubin, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceeding of the European Conference on Software Maintenance and Reengineering, CSMR 2013*, pages 25–34, 2013.

[8] D. Faust and C. Verhoef. Software product line migration and deployment. In *Software Practice and Experience, John Wiley Sons, Ltd*, volume 33, pages 933–955, 2003.

[9] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceeding of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '10*, pages 175–190, 2010.

[10] Mario Luis Guimaraes and Antonio Rito Silva. Improving early detection of software merge conflicts. In *Proceeding of the 35th International Conference on Software Engineering, ICSE 2012*, pages 342–352, 2012.

[11] Jilles Van Gurp and Christian Prehofer. Version management tools as a basis for integrating product derivation and software product families. In *VaMoS 2006*, pages 48–58, 2006.

[12] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceeding of the 15th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2007*, pages 55–64, 2007.

[13] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceeding of the 32nd International Conference on Software Engineering, ICSE 2009*, pages 485–495, 2009.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *IEEE Transactions on Software Engineering*, pages 28(7):654–670, 2002.

[15] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceeding of the 3nd USENIX Symposium on Operating Systems Design and Implementation, OSDI 2004*, pages 151–160, 2004.

[16] Yun Lin, Zhenchang Xing, Yinxing Xue, Xin Peng, Jun Sun, and Wenyun Zhao. Detecting and summarizing differences across multiple instances of code clones. In *Proceeding of the 37th International Conference on Software Engineering, ICSE 2014*, 2014.

[17] Thilo Mende, Felix Beckwermert, Rainer Koschke, and Gerald Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *Proceeding of the 12th European Conference on Software Maintenance and Reengineering, CSMR 2008*, pages 163–172, 2008.

[18] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceeding of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 329–342, 2011.

[19] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits. In *Proceeding of the 36nd International Conference on Software Engineering, ICSE 2013*, pages 502–511, 2013.

[20] M.Gabel and Z.Su. A study of the uniqueness of source code. In *Proceeding of the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2010*, pages 147–156, 2010.

[21] Stas Negara, Mihai Codoban, and Danny Dig. Mining fine-grained code changes to detect unknown change patterns. In *Proceeding of the 37th International Conference on Software Engineering, ICSE 2014*, 2014.

[22] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceeding of the 28th IEEE/ACM International Conference Automated Software Engineering, ASE 2013*, 2013.

[23] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceeding of the 33th International Conference on Software Engineering, ICSE 2010*, pages 315–324, 2010.

[24] Rahul Premraj, Antony Tang, Nico Linssen, Hub Geraats, and Hans van Vliet. To branch or not to branch? In *Proceeding of the International Conference on Software and Systems Process, ICSSP 2011*, pages 81–90, 2011.

[25] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceeding of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE '12*, pages 53:1–53:11, 2012.

[26] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Proceeding of the 28th IEEE/ACM International Conference Automated Software Engineering, ASE 2013*, 2013.

[27] Julia Rubin and Marsha Chechik. Combining related products into product lines. In *Proceeding of the 16th Fundamental Approaches to Software Engineering, FASE 2013*, pages 285–300, 2013.

[28] Julia Rubin and Marsha Chechik. A framework for managing cloned product variants. In *Proceeding of the 36th International Conference on Software Engineering, ICSE 2013*, pages 1233–1236, 2013.

[29] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: A framework and experience. In *Proceeding of the 17th International Software Product Line Conference, SPLC 2013*, pages 101–110, 2013.

[30] Emad Shihab, Christian Bird, and Thomas Zimmermann. The effect of branching strategies on software quality. In *Proceeding of the 6th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2012*, pages 301–310, 2012.

[31] S.Livieri, Y.Higo, M.Matsushita, and K.Inoue. Analysis of the linux kennel evolution using code clone coverage. In *Proceeding of the International Workshop on Mining Software Repositories, MSR 2007*, page 22, 2007.

[32] Mark Staples and Derrick Hill. Experiences adopting software product line development without a product line architecture. In *Proceeding of the 11th Asia-Pacific Software Engineering Conference, APSEC 2004*, pages 176–183, 2004.

[33] `http://blog.boombatower.com/`.

[34] `http://en.wikipedia.org/wiki/Backporting`.

[35] `http://en.wikipedia.org/wiki/Linux_kernel`.

[36] `http://en.wikipedia.org/wiki/Linux_kernel`.

[37] http://en.wikipedia.org/wiki/Rebasing.

[38] http://lwn.net/.

[39] http://lxer.com/.

[40] https://www.kernel.org/category/releases.html.

[41] https://www.kernel.org/doc/Documentation/development-process/2.Process.

[42] https://www.kernel.org/doc/Documentation/stable_kernel_rules.txt.

[43] http://www.engadget.com/2007/10/09/microsoft-backports/.

[44] www.linuxtoday.com/.