A Case Study of Cross-System Porting in Forked Projects

Baishakhi Ray and Miryung Kim
The University of Texas at Austin
Austin, TX USA
rayb@utexas.edu, miryung@ece.utexas.edu

ABSTRACT

Software forking—creating a variant product by copying and modifying an existing product—is often considered an ad hoc, low cost alternative to principled product line development. To maintain such forked products, developers often need to port an existing feature or bug-fix from one product variant to another. As a first step towards assessing whether forking is a sustainable practice, we conduct an in-depth case study of 18 years of the BSD product family history. Our study finds that maintaining forked projects involves significant effort of porting patches from other projects. Cross-system porting happens periodically and the porting rate does not necessarily decrease over time. A significant portion of active developers participate in porting changes from peer projects. Surprisingly, ported changes are less defect-prone than non-ported changes. Our work is the first to comprehensively characterize the temporal, spatial, and developer dimensions of cross-system porting in the BSD family, and our tool REPERTOIRE is the first automated tool for detecting ported edits with high accuracy of 94% precision and 84% recall. Our study finds that the upkeep work of porting changes from peer projects is significant and currently, porting practice seems to heavily depend on developers doing their porting job on time. This result calls for new techniques to automate cross-system porting to reduce the maintenance cost of forked projects.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

software evolution, forking, porting, repetitive changes, code clones

SIGSOFT'12/FSE-20 2012 Cary, NC USA

1. INTRODUCTION

It has become increasingly common to create a variant software product or to introduce a new feature by copying code fragments from similar software products. For example, FreeBSD, OpenBSD, and NetBSD evolved from the same codebase, OpenSSH originated from SSH, LibreOffice originated from OpenOffice.org, etc. As copying code fragments across products is common, there are names referring to this process: *forking*—copying an existing product to create a slightly different product and *porting*—copying an existing feature implementation or bug fix to another member of the same product family. Software forking is often considered an ad hoc, low-cost alternative to principled product line development [26].

Though forking provides flexibility in taking an existing project to new directions or providing software under different license restrictions [8], forking has negative implications during software maintenance. It duplicates development effort and requires developers to port similar bug fixes and feature implications across forked projects [26].

To investigate the extent and characteristics of repeated work in maintaining forked projects, we focus on cross-system porting changes. We compute the amount of edits that are ported from other projects as opposed to the amount of code duplication across projects, because not all code clones across different projects undergo similar changes during evolution, and similar changes are not confined to code clones. For this analysis, we develop a tool called REPERTOIRE that compares the content and edit operations of program patches to identify *ported edits*. REPERTOIRE takes *diff*-based program patches at the release granularity as input. It then uses CCFinderX [15] to identify similar edit content in the patches and determines similar edit operation sequences using *N-gram* matching [1]. To evaluate the accuracy of REPERTOIRE, we manually construct the ground truth of ported edits on a sampled data set. We inspect code changes whose commit messages indicate cross-system porting activities and individual ported edits reported by REPERTOIRE. The comparison between the REPERTOIRE's results against this ground truth finds that it has precision of 94% and recall of 84%.

Using REPERTOIRE, we conduct an in-depth case study of three parallel 18 years of version history of the BSD product family—one of the most well known, long-surviving product family created through software forking. NetBSD and FreeBSD were forked from BSD Lite in 1993 and OpenBSD was forked from NetBSD in 1995. Though they are maintained independently, recent studies indicate that they share

a large amount of common code fragments [9,11,29] and that similar bug fixes are common despite the lack of overlap between contributors [5]. Using the version history of three BSD projects, we investigate the extent of ported changes from other projects, the number of developers who are involved in porting patches, the time taken to port patches, and the locations where ported changes are made to, etc.

Our study questions and findings are summarized as follows:

- **What is the extent of edits ported from other projects?** On average, the amount of edited lines ported from the patches of other projects consists of 13.77%, 15.52%, and 10.74% of total number of lines in the release level patches of FreeBSD, NetBSD, and OpenBSD respectively. Porting patches from other projects happens periodically in the BSD family. The porting rate does not necessarily decrease over time across all three projects.
- **Are ported changes more defect-prone than non-ported changes?** Changes ported from other projects are less defect-prone than non-ported changes in all three projects. This implies that developers are likely to selectively port well-tested features from other projects.
- **How many developers are involved in porting patches from other projects?** In each release, a significant portion of developers port changes from peer projects: on average 26.12%, 58.85%, and 44.85% of active developers in FreeBSD, NetBSD, and OpenBSD respectively. The *entropy* measure of developers is lower for ported changes than non-ported changes, implying that the workload distribution of porting work is skewed: some do a lot more porting than others.
- **How long does it take for a patch to propagate to different projects?** More than 50% of ported edits propagate from one system to another within 10, 13, and 20 months in FreeBSD, NetBSD, and OpenBSD, corresponding to about 2.11, 1.09, and 2.95 releases on average respectively. However, some changes take a very long time to propagate. For 90% of all ported edits to propagate to peer projects, it takes 66, 66, and 81 months.
- **Where is the porting effort focused on?** Ported changes are localized within less than 20% of the modified files per release on average in all three BSD projects. This indicates that porting is concentrated on a few sub systems.

Though the individual BSD development communities have managed to cope with the consequence of forking, the amount of work require'd to port changes from other projects is not insignificant. A considerable amount of time and developer effort is spent on repeated work across forked projects. Currently, the upkeep work of porting changes from other project seems to heavily depend on contributors doing their job on time. These results call for new tool support for notifying relevant developers of potential collateral evolution [25] and propagating a feature implementation or bug fix to relevant contexts in different projects automatically. For example, Sydit [20] and Anderson's approach [3] aim to realize a new means to automatically replicate similar changes and could relieve the burden of cross-system porting of forked projects. A shared change tracking system might also be useful, which will keep track of cross-system porting across forked projects.

Our paper makes the following contributions:

- REPERTOIRE is an automated cross-system porting analysis tool, which finds ported edits with 94% precision and 84% recall. This tool can serve as a basis for assessing the extent and characteristics of cross-system porting among forked projects.
- Our work is the first comprehensive analysis of cross-system porting in the BSD product family along the temporal, spatial, and developer dimensions.
- Our study finds that, while ported edits are less defect-prone than non-ported edits, the upkeep work of cross-system porting is significant and involves a large number of active developers.

As the decision of *forking* has long-term consequences, we plan to further investigate the relationship between porting effort and other software metrics such as dependencies, coupling, people and organization metrics, etc. Since forking decisions are often made due to license disagreement or incompatibility, we aim to investigate the implication of license terms on porting effort or the information flow among forked projects by combining our automated porting analysis with German et al.'s license analysis [11, 12].

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 discusses our study method using an motivating example from the BSD product. Section 4 presents study results. Section 5 discusses threats to validity. Section 6 summarizes the implications of our study results with the directions for future work.

2. RELATED WORK

Studies on Code Duplication. Although most consider code clones to be identical or similar code fragments [15], code clones have no consistent or precise definition in the literature. Indeed, a "clone" has been defined operationally based on the computation of individual clone detectors. Some are based on lexical and syntactic analysis, while others depend on isomorphic program dependence graph analysis, code metrics, etc. A more comprehensive literature on code clones is described elsewhere [28].

By applying clone detection techniques to programs, several studies investigate the extent of clones in software. Nearly as much as 10% to 30% of the code in many large scale projects is identified as clones (e.g., *gcc*-8.7%, *JDK*-29% [15], *Linux*-22.7% etc). Gabel and Su investigate cloning among a collection of 6000 software projects (over 420 million lines of code) and find a general lack of source code uniqueness at the level of approximately one to seven lines [10]. Al-Ekram et al. show that even unrelated software systems have about about 1% of common code, and this accidental cloning is often due to sharing similar API usages, etc [2]. Krinke et al. present an approach of detecting the provenance of code among several projects using clone detection [17]. Livieri et al. study the extent of duplicated code in Linux over time [19].

However, these studies focus on the amount of code duplication in a software system as opposed to the extent of repetitive effort of cross-system porting among forked projects. Not all code clones evolve in the same fashion, and similar changes are not restricted to only cloned code. To measure

the extent of repeated work, REPERTOIRE focuses on similarity among *program patches* across forked software systems as opposed to similarity among code fragments. This requires REPERTOIRE to check whether code fragments undergo similar additions, deletions, and modifications by considering an extra dimension of *edit operation* similarity.

Case Studies on the BSD Product Family. Several studies analyzed the evolution of BSD product family. For example, Fischer et al. analyzed change commit messages of the BSD family and found a decreasing trend of information flow between OpenBSD and other BSD projects [9]. Their analysis does not automatically identify similar code modifications made to different BSD projects. Yamamoto et al. found up to 40% of lines of code are shared among NetBSD, OpenBSD, and FreeBSD [29]. James et al. showed the evidence of adopted code in device driver modules between Linux and FreeBSD [7]. German et al. also studied cross project cloning in the BSD product family and analyzed copyright implications when code fragments transfer between different systems under different licenses. On the other hand, our study focuses on the characteristics of cross-system ported changes. Canfora et al. investigated the social characteristics of contributors who make cross-system bug fixes between FreeBSD and OpenBSD [5]. They used textual analysis of change commit logs and mailing list communication logs. Their findings are aligned with our finding that contributors who port changes from other projects are highly active contributors. Unlike Canfora et al., our study investigates all three projects (OpenBSD, FreeBSD, and NetBSD), and automatically detects ported changes by determining similar edit content and edit operation sequences within release-level patches as opposed to the textual analysis of change commit messages only. Furthermore, our study extends Canfora et al. by measuring the time taken to port changes, the percentage of files affected by porting, the workload distribution among the contributors who port patches from other projects, and the correlation between defects and ported changes vs. non-ported changes. Ozment et al. investigated security vulnerabilities in the OpenBSD project to examine whether software security improves with age [24]. However, they did not investigate the extent and frequency of ported changes from other BSD projects.

Clone Evolution Analysis. Lague et al. first analyzed the evolution of clones over time in a large telecommunication system and classified changes to code clones in four categories: new, modified, never modified, and deleted [18]. Kim et al. developed clone genealogy analysis to study changes to code clones [16]. Balint et al. developed a visualization tool to show (1) who created and modified code clones, (2) the time of the modifications, (3) the location of clones in the system, and (4) the size of code clones [4]. These studies detect code clones a priori in software systems and monitor changes to only those clones over time. In contrast to clone evolution analysis, our analysis compares the content and edit sequences of *program patches* to detect ported changes among forked BSD projects.

Recurring Software Modifications. Previous work on recurring bug fixes [23] finds that a large number of similar bug fixes are made to *code peers*, which provide similar functionality or use APIs in a similar manner. Their empirical analysis focuses on recurring bug fixes and security vulnerabilities in individual projects but does not investigate cross-system bug fixes in a product family. Padioleau et al. [25] investigate the extent of recurring software modifications among device drivers in Linux. Based on the insight that making similar changes to not identical contexts is tedious and error-prone, several approaches infer a generalized program transformation script from example program differences to automate similar edits [3, 20]. Our study provides the motivation for applying such approaches to facilitate collateral evolution of forked software projects.

3. STUDY METHOD

Sections 3.1 and 3.2 describe our study subjects and our tool REPERTOIRE that automatically identifies ported edits within program patches. Section 3.3 describes how we measure the accuracy of REPERTOIRE through a manual inspection of change logs and program patches and how we tune the input threshold for CCFinderX for our study.

3.1 Study Subjects

For our case study, we focus on the three BSD projects, which share a common ancestor. While OpenBSD was directly forked from NetBSD, FreeBSD and NetBSD were forked from a common origin BSD Lite. We use 54, 14, and 30 releases from FreeBSD, NetBSD, and OpenBSD and thus covering 18 years of parallel evolution history. Table 1 shows the size of each BSD, the releases studied, and the number of developers in each project.

Since all three BSD projects under consideration use a CVS repository, we use `cvs diff` to identify program patches applied to individual projects, use `cvs log` to identify commit message, and use `cvs annotate` to retrieve committer information. To identify bug fixes for each project, we parse each file's change commit messages and identify versions that contain keywords such as *'patch,' 'fix,'* and *'bug,'* using a heuristic developed by Mockus and Votta [21].

Table 1: The BSD Product Family

	KLOC	releases	authors	years
FreeBSD	359 to 4479	54 (R1.0 - R8.2)	405	18
NetBSD	859 to 4463	14 (R1.0 - R5.1)	331	18
OpenBSD	297 to 2097	30 (R1.1 - R5.0)	264	16

3.2 Repertoire

To detect ported edits within individual program patches, REPERTOIRE determines similar edit content and operations between each pair of patches. By program patches, we mean *diff*-based line-level differences per file. We focus our attention on `.c` files only, discarding header files, because we are interested in changes to implementations rather interface declarations. The *diff*-based patches are generated using a `cvs diff` command at the release granularity. We also use the `-c` option to include surrounding unchanged code and use the `-p` option to organize program differences per function.

REPERTOIRE identifies similar program modifications between patches in the following three steps. Consider the two input patches P_x and P_y shown in Table 2.

① Identify cloned regions between patches. First, REPERTOIRE pre-processes *diff*-based patches to convert them into a CCFinderX compatible format. It removes symbols

Table 2: REPERTOIRE compares both the content and edit operations of patches.

P_x

```
**** Old ****
X1  for(i=0;i<MAX;i++){
X2  !  x = array[i]  + x;
X3  !  y = foo(x);
X4  -  x = x - y;
X5  }
**** New ****
X6  for(i=0;i<MAX;i++) {
X7  +  y = x + y;
X8  !  x = array[i] + x;
X9  !  y = foo(x,y);
X10 }
```

P_y

```
**** Old ****
Y1   for(j=0;j<MAX;j++) {
Y2     q = p + q;
Y3  !  q = array[j] + p;
Y4  !  p = foo1(q);
Y5  }
**** New ****
Y6   for(j=0;j<MAX;j++) {
Y7     q = p + q;
Y8  !  q = array[j] + q;
Y9  !  p = foo1(p,q);
Y10 }
```

representing edit operation types, such as + for added lines, - for deleted lines, and ! for modified lines. It also removes *diff* specific meta information such as a revision number, a modification date, etc. By running CCFinderX [15] on the pre-processed patches, it finds a set of cloned region pairs across the input patches.

REPERTOIRE currently uses a minimum token threshold, 40 tokens for CCFinderX, because using 40 tokens led to the best precision and recall among the threshold values that we tested. In the next section, we describe how we select a minimum token threshold for CCFinderX through an accuracy evaluation of REPERTOIRE in detail.

REPERTOIRE then removes cloned region pairs between the old context and the new context of patches, because these pairs represent cloning relations between *deleted* lines in the old context and *added* lines in the new context, and thus do not exhibit similar program modifications. For example, given the two patches, P_x and P_y, CCFinderX finds three pairs of cloned regions: (lines X2 to X3, lines Y3 to Y4), (lines X6 to X10, lines Y6 to Y10) and (lines X7 to X8, lines Y2 to Y3). Then REPERTOIRE removes a pair of cloned regions, (lines X7 to X8, lines Y2 to Y3), from the results because it does not represent similar edits.

② Retrieve edit operation sequences from the cloned regions. REPERTOIRE then identifies edit operation sequences for the cloned regions in the previous step. For example, REPERTOIRE retrieves edit operation sequences for each cloned region, lines X2 to X3, lines Y3 to Y4, lines X6 to X10, and lines Y6 to Y10. For example, lines X2 to X3 produces a sequence of two modifications, noted as `'!!'`. Lines X6 to X10 produces a sequence of three edit operations, noted as `'+!!'` because X6 and X10 are unchanged lines. Lines Y3 to Y4 produces a sequence of two modifications, noted as `'!!'`. Lines Y6 to Y10 produces a sequence of two modifications, noted as `'!!'` because Y6, Y7, and Y10 are unchanged lines.

③ Identify similar edit operation sequences using the bi-gram matching. To find similar edit sequences, REPERTOIRE uses the bi-gram matching algorithm [1]. We use a bi-gram matching instead of the longest common subsequence algorithm [14], because the bi-gram matching could allow slight variations in edit sequences. When matching edit operations, we match added lines (+) with modified lines (!) of a patch's new context, because they have the same effect. Similarly, we match deleted lines (-) with modified lines (!) of a patch's old context. As a result of bi-gram matching, REPERTOIRE finds similar program modifications between the two patches P_x and P_y: similar deletions are made to lines X2 to X3 and lines Y3 to Y4. Similar additions are made to lines X8 to X9 and lines Y8 to Y9.

Table 3 shows an example of ported edits found by REPERTOIRE. The corresponding change logs for FreeBSD and NetBSD show that the device support for RTL8211C(L) was ported from FreeBSD to NetBSD. Note that the old code fragments in the two projects are not exactly clones of each other according to CCFinderX, though both code fragments experienced similar program modifications. This example highlights that detecting ported edits requires finding similar *edits* as opposed to finding similar *code fragments* a priori and monitoring edits to only the found clones.

3.3 Accuracy Evaluation

To assess REPERTOIRE's accuracy in detecting ported edits, we manually construct a ground truth of ported edits on a sampled evolution period of OpenBSD releases 4.4 to 4.5. To create the ground truth, we collect candidate ported edits using the following two methods.

First, we extract program revisions whose change comments indicate porting from NetBSD to OpenBSD. From the CVS history of 11/1/2008 to 5/1/2009, which corresponds to 4.4 and 4.5 release dates, we search for keywords 'NetBSD' or 'NETBSD' in the check-in messages. For example, we find file revision, `src/sys/compat/ultrix/ultrix_misc.c:v 1.31` with the message *'Make ELF platforms generate ELF core dumps. Somewhat based on code from NetBSD.'*

Second, we run REPERTOIRE between the OpenBSD patch from 4.4 to 4.5 and all preceding 12 release-level patches in NetBSD up to release 4.0 using a very low token threshold, 20 tokens. By setting the token threshold to a very small number, REPERTOIRE over-approximates potential ported edits. We then merge candidate ported edits from two different sources and removed false positive edits by manually inspecting *diff* outputs and commit messages. As a result, we construct the ground truth of ported edits at a line granularity for OpenBSD release 4.5: total 1429 lines of edits are ported from NetBSD patches and these edits span across 90 files.

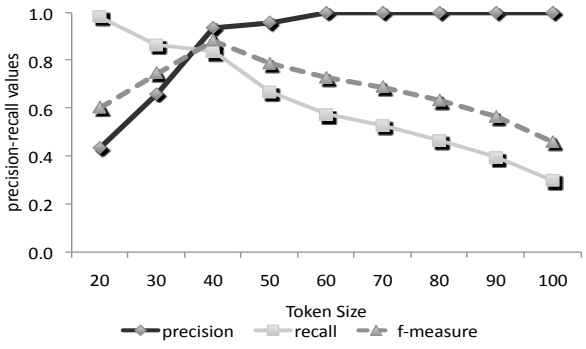


Figure 1: Precision and recall values of ported edits found by REPERTOIRE while varying the token threshold values from 20 to 100 tokens

We then compare the output of REPERTOIRE against this ground truth to measure its precision and recall, which are defined as follows. Suppose that E denotes our ground truth, and R represents the result of REPERTOIRE.

Table 3: An example of ported edits found by REPERTOIRE. Ported edits are colored in gray

A segment of FreeBSD patch

Location: `src/sys/dev/mii/rgephy.c; revision 1.20`

Change Log: Add RTL8211C(L) support. Disable advanced link-down power saving in phy reset

Date: 2008/07/02

```
--- 531,548 ----
531.    static void
532.    rgephy_reset(struct mii_softc *sc)
533.    {
534.  +  struct rgephy_softc *rsc;
535.  +  uint16_t ssr;
536.  +
537.  +  rsc = (struct rgephy_softc *)sc;
538.  +  if (rsc->mii_revision == 3) {
539.  +    /* RTL8211C(L) */
540.  +    ssr = PHY_READ(sc, RGEPHY_MII_SSR);
541.  +    if ((ssr & RGEPHY_SSR_ALDPS) != 0) {
542.  +      ssr &= ~RGEPHY_SSR_ALDPS;
543.  +      PHY_WRITE(sc, RGEPHY_MII_SSR, ssr);
544.  +    }
545.  +  }
546.
547.     mii_phy_reset(sc);
548.     DELAY(1000);
```

A segment of NetBSD patch

Location: `src/sys/dev/mii/rgephy.c; revision 1.23`

Change Log: Support for RTL8211C(L) phy from FreeBSD

Date: 2009/01/09

```
--- 583,604 ----
583. rgephy_reset(struct mii_softc *sc)
584.  {
585.    struct rgephy_softc *rsc;
586.  + uint16_t ssr;
587.
588.    mii_phy_reset(sc);
589.    DELAY(1000);
590.
591.    rsc = (struct rgephy_softc *)sc;
592.  + if (rsc->mii_revision < 2) {
593.      rgephy_load_dspcode(sc);
594.  + } else if (rsc->mii_revision == 3) {
595.  +     /* RTL8211C(L) */
596.  +     ssr = PHY_READ(sc, RGEPHY_MII_SSR);
597.  +     if ((ssr & RGEPHY_SSR_ALDPS) != 0) {
598.  +       ssr &= ~RGEPHY_SSR_ALDPS;
599.  +       PHY_WRITE(sc, RGEPHY_MII_SSR, ssr);
600.  +     }
601.  + } else {
602.
603.    PHY_WRITE(sc, 0x1F, 0x0000);
604.    PHY_WRITE(sc, 0x0e, 0x0000);
```

Table 4: Examples of a false positive and a false negative

	Date	Project	Committer	ChangeLog
FP	1999/03/26	NetBSD	bouyer	`src/usr.bin/eject/eject.c`: Oups, complete braindamage yesterday. DIOCEJECT does the rigth thing for both disks and CDs, it's just don't have to call DIOCLOCK before, unless we're doing a forced eject: DIOCEJECT will check for device use and unlock the door if allowed.
	2008/07/23	OpenBSD	djm	`src/usr.bin/ssh/servconf.c` : do not try to print options that have been compile-time disabled in config test mode (sshd -T); report from nix-corp AT esperi.org.uk ok dtucker@
FN	2009/01/29	OpenBSD	thib	`src/sys/nfs/nfs_bio.c` : Use a timespec instead of a time_t for the clients nfsnode mtime, gives us better granularity, helps with cache consistency.Idea lifted from **NetBSD**.

Table 5: Sample ported edits found by REPERTOIRE.

	Date	Project	Committer	ChangeLog
1.	1997/04/23	NetBSD	scottr	Implement new crash dump format. Mostly taken from hp300, extended to support multiple physical RAM segments by me. Garbage collect functions obsoleted by this change.
	1999/04/23	OpenBSD	downsj	Kcore dump, from **NetBSD**.
2.	2002/03/01	OpenBSD	espie	Kill hand-made memory allocation code that is definitely buggy. Replace with simple wrapper around malloc, at least this works, and it's easier to debug anyways.
	2004/07/07	NetBSD	mycroft	Cleanup of ksh memory handling from **OpenBSD** via Stefan Krueger in PR 24962.
3.	2006/09/09	FreeBSD	ambrisko	Add support to bge(4) to not break IPMI support when the driver attaches to it. Try to co-operate with the IPMIASF firmware accessing the PHY
	2010/01/28	NetBSD	msaitoh	Introduce IPMI and ASF related code from **FreeBSD.**
4.	2009/07/03	OpenBSD	dlg	this is a rather large change to add support for the BCM5709.
	2010/01/27	NetBSD	sborrill	Add support for the Broadcom BCM5709 and BCM5716 chips

Precision: the percentage of ported lines found by REPERTOIRE that are also present in the ground truth, i.e., $\frac{|E \cap R|}{|R|}$

Recall: the percentage of the ground truth that is also present in the REPERTOIRE's results, i.e., $\frac{|R \cap E|}{|E|}$

To select a token threshold setting for CCFinderX, we then vary the token size from 20 to 100 tokens in increment of 10 and measure the accuracy of REPERTOIRE. Our accuracy evaluation finds that, at token size 40, the precision value is 0.94 and the recall value is 0.84. The values are shown in Figure 1. The F-measure is defined as a harmonic mean of precision and recall and it reaches a maximum value of 0.88 at token size 40. We use this threshold of 40 tokens throughout the empirical study in Section 4.

Table 4 shows examples of a false positive and a false negative reported by Repertoire, when using a token threshold 40 for CCFinderX. In the case of the false positive, Repertoire detects ported edits between the two patches, though there is no semantic similarity between surrounding contexts. Such false positive was found because false positive clones could be found by CCFinderX. In the case of the false negative, Repertoire was not able to detect ported edits, because the contiguous lines of ported edits are less than 40 tokens long.

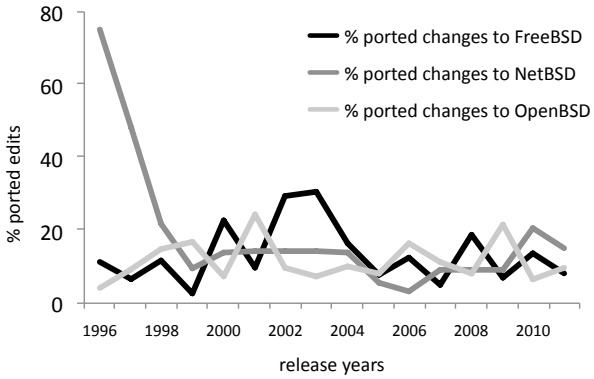
Table 5 shows few samples of the positive ported edits between the three BSD projects. As shown in the corresponding commit messages, BSD developers port bug fixes and new features from peer projects. Among the four examples, examples 1 through 3 show ported edits that are found by REPERTOIRE and validated by their respective change logs. The 4th example is found by REPERTOIRE but there is no explicit mention of other projects in the commit messages. Such example shows that commit messages alone are inadequate for identifying ported edits and highlights the benefit of using an automated tool like REPERTOIRE for an empirical study of cross-system porting in the BSD family.

4. STUDY RESULTS

This section describes the characteristics of ported code changes in the BSD family. Section 4.1 describes the extent and frequency of ported changes. Section 4.2 compares defect density of ported changes against non-ported changes. Section 4.3 describes the work load distribution of developers who port changes from other projects. Section 4.4 describes the time taken to port patches from other projects, and Section 4.5 describes the code locations where ported changes were made to.

4.1 What is the extent of changes ported from other projects?

We analyze the extent of ported changes for individual projects by comparing program patches at a release granularity. For example, to measure the percentage of NetBSD changes originated from OpenBSD and FreeBSD, we compute program patches for all NetBSD releases. A NetBSD patch $\Delta NetBSD_{(i-1,i)}$ is generated using `cvs diff` between release i and its prior release $i-1$. We then list all patches created in the peer projects prior to the release date of NetBSD release i. Based on the assumption that the code changes made in the peer projects must be available first to be transferred to another project, we compare these patches with $\Delta NetBSD_{(i-1,i)}$ using REPERTOIRE and identify the number of code lines ported from peer projects in each patch.



		Free Only	Net Only	Open Only	Both	Total
FreeBSD	AVG		3.19%	8.36%	2.22%	13.77%
	MED		0.94%	5.74%	1.49%	10.78%
NetBSD	AVG	2.71%		7.87%	4.94%	15.52%
	MED	1.31%		4.65%	3.08%	14.34%
OpenBSD	AVG	4.61%	4.20%		1.93%	10.74%
	MED	3.34%	3.30%		1.64%	9.52%

Each row represents a target project, each column represents a source project.

Figure 2: The porting rates in the BSD family

The porting rate in each release is computed as the percentage of line additions and deletions ported from other projects out of the total number of line additions and deletions in the patch. For example, for Table 3, the porting rate would be 80% because there are 10 line additions in the NetBSD patch and REPERTOIRE finds that 8 out of them are ported from FreeBSD. We calculate the average porting rate across all releases of a project as:

$$avg.\ porting\ rate = \frac{\sum_{releases} ported\ edits}{\sum_{releases} total\ edits}$$

The porting rate of NetBSD across 15 releases ranges from 3.25% to 75.16%. The average number of ported line additions and deletions per NetBSD release is 45,429 CLOC (changed LOC) and the average size of NetBSD patch is 292,667 CLOC, producing an average porting rate of 15.52%. On average ported edits are 12,127 out of 88,053 CLOC in FreeBSD and 16,927 out of 157,612 CLOC in OpenBSD, resulting in an average porting rate of 13.77% and 10.74% respectively. Figure 2 shows average porting rates for individual projects and their median values. Some ported edits are from one project only while other ported edits are found from the patches of both projects. For example, out of 13.77% of ported edits in FreeBSD patches, 3.19% comes from NetBSD patches only, 8.36% comes from OpenBSD patches only, and the rest 2.22% is found in both NetBSD and OpenBSD patches. In all three projects, the median value is lower than the average value. In most releases, the amount of ported edits is lower than the average, while in some releases, ported edits consist of a significant portion of individual patches. In the BSD family, porting is a periodic phenomenon.

	(1996 - 2000)			(2000 - 2011)		
	m	c	p-value	m	c	p-value
Free	1.89	5.37	0.51	-1.27	22.12	0.13
Net	-16.03	82.09	0.03	0.08	11.54	0.87
Open	1.42	6.38	0.48	-0.39	14.54	0.53

To understand porting rate changes, we apply linear regression on the data set of Figure 2. The results are in the form of $y = mx + c$ where y is a porting rate and x is a release year and are shown in the table above. The porting rate since year 1996 does not necessarily decrease over time in FreeBSD and OpenBSD. The linear regression analysis of porting rates since year 2000 shows no negative m values, where p-value < 0.05.

Porting consists of a significant portion of the BSD family evolution and porting rates do not necessarily decrease over time. These results call for new tool support for notifying relevant developers of potential collateral evolution and propagating the changes automatically.

Porting consists of a significant portion of the BSD family evolution, corresponding to 14%, 16%, and 11% porting rates on average in FreeBSD, NetBSD, and OpenBSD.

4.2 Are ported changes more defect-prone than non-ported changes?

To investigate the relationship between bug fixes and ported changes, we first identify all bug fixes made to individual BSD projects by searching for keywords, *'bug,' 'fix,'* and *'patch'* in change commit messages using a heuristic similar to Mockus and Votta [21]. For each file, we then measure the cumulative number of changed lines (CLOC), ported lines (Ported CLOC), and non-ported lines (Non-Ported CLOC) over all releases using the results of ported edits found by Repertoire. We consider only source files and exclude header and configuration files. For example, in total, 4,754,862 line additions and deletions are made in FreeBSD over the study period, out of which 654,858 lines are identified as ported edits and 4,100,004 lines are non-ported edits.

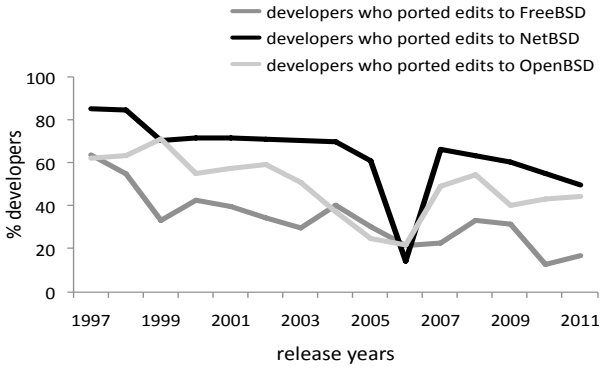
We then measure the Spearman rank correlation between the number of bug fixes and ported CLOC at the file granularity [30]. Similarly, we measure the correlation between bug fixes and non-ported CLOC. We use a *rank correlation* to control for *churn*, in other words, the total number of added and deleted lines. In all three projects, the correlation between bug fixes and ported edits is weaker than the correlation between bug fixes and non-ported edits: 0.15 (ported) vs. 0.25 (not-ported) in FreeBSD, 0.36 vs. 0.42 in NetBSD, and 0.32 vs. 0.38 in OpenBSD. These correlations are statistically significant with p-values $<$ 2.2e-16. In all three projects, the correlation between *churn* and bugs is higher than the correlation between ported edits and bugs. While code churn is highly correlated with defects and is a good predictor of bugs [22], our results indicate that *ported churn is likely to be relatively more safe and reliable than non-ported churn.*

These results indicate that developers port well-tested bug fixes and feature implementations from peer projects rather than risky and experimental features. This benefit of selective porting is aligned with Cordy's observation. In financial software industry, copying an existing product to create a variant product is a recommended practice [6], because software forking allows developers to independently evolve product variants and reduces the risk of collective system failures caused by using a common platform.

Table 6: Spearman rank correlation between bug fixes and ported changes vs. non-ported changes

	CLOC	Ported CLOC	Non-Ported CLOC
FreeBSD	4754862	654858	4100004
Correlation with bugs	0.26	0.15	0.25
p-value	$<$ 2.2e-16	$<$ 2.2e-16	$<$ 2.2e-16
NetBSD	4097338	636006	3461332
Correlation with bugs	0.41	0.36	0.42
p-value	$<$ 2.2e-16	$<$ 2.2e-16	$<$ 2.2e-16
OpenBSD	4728360	507810	4220550
Correlation with bugs	0.37	0.32	0.38
p-value	$<$ 2.2e-16	$<$2.2e-16	$<$ 2.2e-16

Files with ported edits are less defect-prone than the files with non-ported edits. This indicates that developers may selectively port well-tested features and bug fixes from peer projects.



		Free Only	Net Only	Open Only	Both	Total
FreeBSD	AVG		6.65%	5.52%	13.95%	26.12%
	MED		4.63%	5.47%	14.05%	25.18%
NetBSD	AVG	5.73%		20.13%	32.99%	58.85%
	MED	3.83%		20.18%	43.45%	68.19%
OpenBSD	AVG	6.40%	12.89%		25.56%	44.85%
	MED	6.71%	12.79%		26.56%	45.53%

Each row represents a target project, each column represents a source project.

Figure 3: The percentage of developers who port changes from other projects per release

4.3 How many developers are involved in porting patches from other projects?

We hypothesize that the maintenance cost of forked projects is high and the porting effort is prevalent, if it involves a large percentage of development communities. To investigate this hypothesis, we identify developers who committed ported edits using `cvs annotate` and compute the total number of those developers in each release. For release i, the percentage of developers involved in porting is defined as the

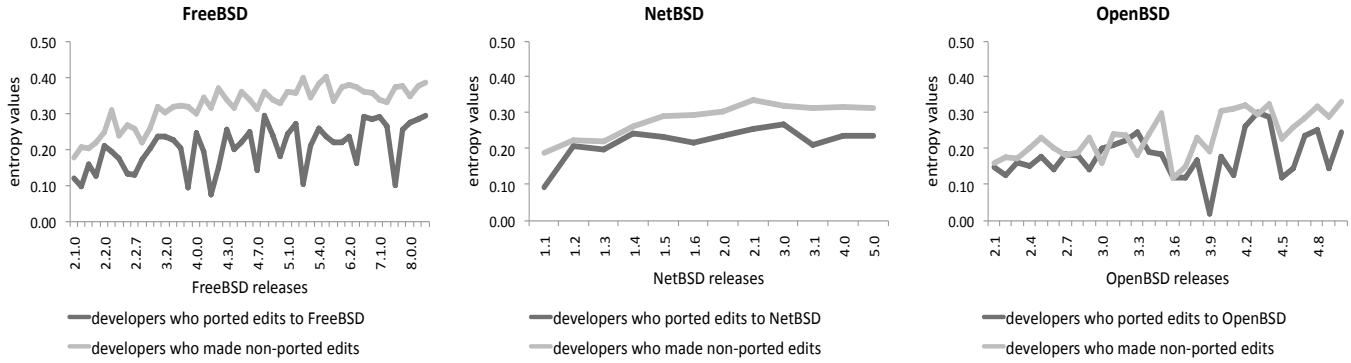


Figure 4: The workload distribution of developers of ported changes vs. non-ported changes in terms of entropy. X axis represent releases for each project.

ratio of the number of developers who ported edits in release i to the total number of active contributors of release i. Figure 3 shows the average percentage of developers who port changes per release. On average, 26.12% (38 out of 145), 58.85% (91 out of 155), and 44.85% (43 out of 96) of committers are involved in porting changes from peer projects per release in FreeBSD, NetBSD, and OpenBSD. Out of all active developers, around 13.95%, 32.99% and 25.56% port changes from both the other two projects.

To investigate the work load distribution among the developers who port changes from peer projects, we calculate a normalized entropy score of developer contribution. Entropy is a well-known measure of uncertainty [27]. A normalized static entropy is used by Hassan et al. to account for the varying number of active units over time (the number of active developers in our case vs. the number of modified files in Hassan's case) [13] and is defined as follows:

$$normalized\ entropy = -\sum_{i=1}^{n} p_i * log_n(p_i)$$

where p_i is the probability of a line modification that belongs to author i, when there are n unique active developers. We compute this entropy score for each release. A low entropy score implies that only a few developers make most of the modifications. If the entropy is high, it implies that the work load is more equally distributed among the contributors. Figure 4 shows that the entropy measure of ported edits vs. non-ported edits over all releases. The dark gray line (the developer entropy of ported changes) stays below the light gray line (the developer entropy of non-ported changes) in all three projects. The work load distribution is more skewed for ported changes than non-ported changes, implying that some do much more porting work than others.

A significant portion of active committers port changes from other projects.

4.4 How long does it take for a patch to propagate to different projects?

We investigate how long it takes for a patch to propagate from one project to another. We measure the difference between the release date of a source patch and the release date of a target patch for each ported line. We then calculate the average days to propagate a patch, which is defined as follows:

$$porting\ time = \frac{\sum_{r=1}^{N}\sum_{l}^{L(r)} days\ to\ port\ l\ in\ release\ r}{\sum_{r=1}^{N} L(r)}$$

where $L(r)$ is the total number of ported lines of code in release r and N is the total number of releases in a project. It takes on average 734, 725, and 944 days to port an edit from other projects to FreeBSD, NetBSD, and OpenBSD respectively. Figure 5 shows a cumulative distribution of ported edits vs. propagation time in months. On average 50% of ported changes migrate within 10, 13, and 20 months in FreeBSD, NetBSD, and OpenBSD respectively, which correspond to 2.11, 1.09, and 2.95 releases when we map the propagation time to the number of releases. However, some changes take a very long time to propagate. For 90% of all ported changes to migrate, it takes 66 months (19 out of 54 releases) in FreeBSD, 66 months (5 out of 12 releases) in NetBSD and 81 months (17 out of 33 releases) in OpenBSD.

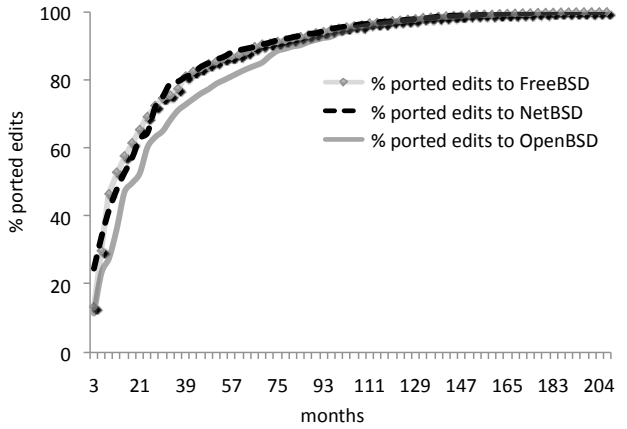


Figure 5: The cumulative distribution of ported changes from other projects vs. patch propagation time.

Though individual BSD projects mostly have managed to keep up-to-date with porting features and bug fixes from

other projects, some changes still take a very long time to be incorporated by other projects.

While most ported changes migrate to peer projects in a relatively short amount of time, some changes take a very long time to propagate to other projects.

4.5 Where is the porting effort focused on?

If ported edits are spread throughout the codebase, we could conclude that the developers who port changes from other projects may need to spend a significant amount of time, gaining expertise on different parts of the codebase. To investigate where ported edits are made, we measure the file level distribution of ported changes in each BSD project. We consider a file to be affected by porting in the i^{th} release, if it is modified by at least one ported edit since release $i-1$. We define the ratio of files edited by porting in the i^{th} release as the number files with ported edits in release i divided by the total number of edited files in release i. Figure 6 shows the average percentage of files with ported changes. On average, ported changes touch 11.58% of all modified files in FreeBSD, 18.62% in NetBSD, and 15.86% in OpenBSD. A linear regression on the data-sets of Figure 6 shows that the ratio of files modified affected by ported edits is decreasing over time. In the table below, the results are in the form of $y = mx + c$, where y is the percentage of edited files affected by porting and x is a release year.

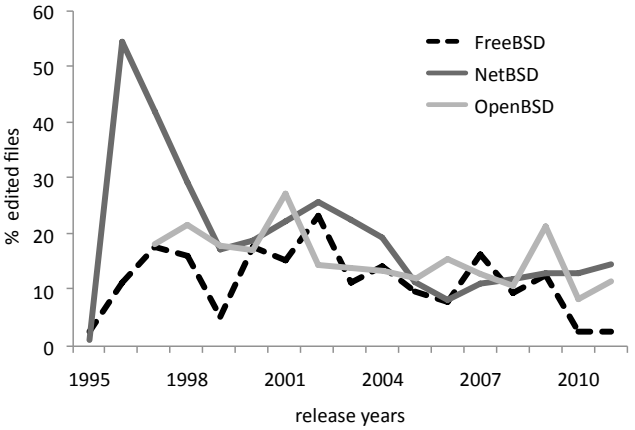
	m	c	p-value
FreeBSD	-0.32	14.46	0.30
NetBSD	-1.29	31.46	0.04
OpenBSD	-0.64	22.21	0.02

These results indicate that porting in the BSD projects is mostly a localized phenomenon. To further understand where this porting effort is focused on, we also calculate the total number of ported lines over all releases in each file. We rank the files in terms of ported edits for the entire study period. Table 7 shows top 10 sub-directories in each project with the highest number of ported lines. These results indicate that porting is localized to a few sub-directories. For example, 21.54% of total porting over the entire study period occurred in `openssl` sub-directory for FreeBSD, 20.34% in `arch` sub-system for NetBSD and 24.57% in device-driver for OpenBSD. In fact, for all three BSD projects, most of the porting efforts are concentrated on the device drivers, crypto APIs, networking services, SSL (secure socket layer) related features, etc.

Ported changes affect about 12% to 19% of modified files and porting effort is concentrated on specific parts of the BSD codebase.

5. THREATS TO VALIDITY

Threats to *construct validity* concern the relation between theory and observation. We rely on the effectiveness of the widely used clone detector CCFinderX to identify similar edit contents among patches. To limit the presence of false positives, we restrict our focus on substantially similar edit contents—at least 40 tokens long—and consider the extra



		Free Only	Net Only	Open Only	Both	Total
FreeBSD	AVG		2.96%	5.30%	3.32%	11.58%
	MED		2.23%	3.96%	3.45%	11.35%
NetBSD	AVG	2.98%		11.53%	4.11%	18.62%
	MED	2.25%		8.11%	4.04%	15.19%
OpenBSD	AVG	5.66%	6.63%		3.57%	15.86%
	MED	4.84%	5.64%		3.30%	14.45%

Each row represents a target project, each column represents a source project.

Figure 6: The percentage of edited files due to porting

dimension of matching the edit operation type in addition to patch content similarity. Thus, lines within patches are considered similar, *only if* both their contents and operations are similar. Furthermore, we evaluate the accuracy of REPERTOIRE by comparing its results against the manually created, ground truth of ported edits on a sampled release patch (OpenBSD 4.5). In order to facilitate the replication of our study, we make our tool and results available at `http://dolphin.ece.utexas.edu/Repertoire.html`.

In terms of temporal granularity, we use program patches between each consecutive release pair; thus, our study cannot detect ported edits, which are once made but reverted prior to the next release. When preparing patches using `cvs diff`, we limit unchanged lines before and after each changed block up to three lines—we speculate the amount of context lines does not affect our result, because those unchanged lines are not counted as ported edits by REPERTOIRE.

In terms of threats to *internal validity*, it is possible that a weak correlation between ported changes and bug fixes is caused by different factors, such as the expertise level of developers who work on subsystems where porting is frequent. Our findings in Section 4.2 indicate *only* correlation with defect density *not* causation.

External validity concerns the generalization of the findings. Our study focuses on FreeBSD, NetBSD and OpenBSD. While we acknowledge that our case study on BSD may not generalize to other systems, we argue that our results on the BSD family are meaningful—the BSD product family is a long-surviving, very large product family, created by software forking and our study findings generate a set of specific hypotheses to be tested in other forked projects such as OpenSSH and SSH, MariaDB and MySQL, LibreOffice and OpenOffice, and various distributions of Linux. We hope that other researchers replicate our results and thereby al-

Table 7: Top ten directories in individual BSD projects with the largest amount of ported changes. The % values are the ratios of ported edits in the individual directories to all ported changes.

Rank	FreeBSD		NetBSD		OpenBSD	
1	src/crypto/openssl	21.54%	src/sys/arch	20.34%	src/sys/dev	24.57%
2	src/crypto/openssh	13.98%	src/sys/dev	19.96%	src/lib/libssl	16.36%
3	src/crypto/heimdal	13.31%	src/crypto/dist	10.61%	src/sys/arch	11.16%
4	src/sys/dev	8.95%	src/gnu/dist	4.54%	src/usr.sbin/ppp	6.27%
5	src/sys/contrib	5.26%	src/sys/netinet	3.08%	src/gnu/usr.bin	5.27%
6	src/lib/libc	3.08%	src/lib/libc	2.81%	src/sys/netinet	2.93%
7	src/usr.sbin/ppp	2.56%	src/sys/netinet6	2.66%	src/kerberosV/src	2.71%
8	src/gnu/usr.bin	1.93%	src/sys/kern	2.56%	src/lib/libc	2.31%
9	src/usr.sbin/pppd	1.59%	src/sys/nfs	2.27%	src/usr.bin/less	1.72%
10	src/sys/nfs	1.46%	src/sys/dist	1.84%	src/sys/kern	1.69%

low the community to build an empirical body of knowledge on the impact of forking and porting on various aspects like quality, dependencies, etc.

6. CONCLUSION AND FUTURE WORK

Software forking is considered an ad-hoc, low-cost alternative to principled product line development. Forking has negative connotations because it requires developers to port similar features and bug fixes from peer projects during software evolution. As a first step toward understanding the longitudinal impact of forking on maintainability and assessing whether forking is a sustainable practice, we developed an automated cross-system porting analysis tool, called Repertoire.

By applying Repertoire to 18 years of parallel release history of the BSD product family, we conducted an in-depth case study of BSD projects. Our study found that the maintenance effort of cross-system porting is significant. About 10.74% to 15.52% of lines in BSD release patches consist of ported edits. 26.12% to 58.85% of active developers participate in cross-system porting per release on average. These results together indicate that, while forking has some benefit of allowing independent evolution, the cost of cross-system porting is significant. Our study is also the first to find that ported changes are likely to be more reliable than non-ported changes, showing the benefit of selectively porting well-tested features. Furthermore, our study found that over 50% of ported changes propagate to other projects within 3 releases, while some changes take a very long time to propagate. Currently, cross-system porting in the BSD community seems to heavily depend on developers doing their porting job on time.

Our results call for an automated approach of applying similar program transformations to related contexts among forked projects or notifying developers of potential collateral evolution. To guide design of such approaches, we plan to further study the types of adaptations required to port changes across peer projects. In terms of future work, we plan to understand the implication of license terms on porting effort or information flow among a group of related projects by combining our automated analysis with license analyses by German et al. [11, 12].

Acknowledgements

We thank Jihun Park for gathering the bug history data for FreeBSD, NetBSD, and OpenBSD projects. This work was in part supported by National Science Foundation under the grants CCF-1117902, CCF-1149391, and CCF-1043810 and by a Microsoft SEIF award.

7. REFERENCES

[1] G. W. Adamson and J. Boreham. The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information Storage and Retrieval*, 10(7-8):253–260, 1974.

[2] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *ISESE '05: International Symposium on Empirical Software Engineering*, pages 376–385, 2005.

[3] J. Andersen and J. Lawall. Generic patch inference. In *ASE '08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 337–346, 2008.

[4] M. Balint, R. Marinescu, and T. Girba. How developers copy. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 56–68, 2006.

[5] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta. Social interactions around cross-system bug fixings: the case of freebsd and openbsd. In *MSR '11: Proceeding of the 8th working conference on Mining software repositories*, pages 143–152, 2011.

[6] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 196–205, 2003.

[7] J. R. Cordy. Exploring large-scale system similarity using incremental clone detection and live scatterplots. In *ICPC '11: 19th IEEE International Conference on Program Comprehension*, pages 151 –160, 2011.

[8] M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 145–154, 2010.

[9] M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall. Mining evolution data of a product family. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5. ACM, 2005.

[10] M. Gabel and Z. Su. A study of the uniqueness of source code. In *FSE '10: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[11] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *MSR '09: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90, 2009.

[12] D. M. German and A. E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 188–198. IEEE Computer Society, 2009.

[13] A. Hassan. Predicting faults using the complexity of code changes. In *ICSE '09: IEEE 31st International Conference on Software Engineering*, pages 78 –88, 2009.

[14] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

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

[16] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2005.

[17] J. Krinke, N. Gold, Y. Jia, and D. Binkley. Cloning and copying between gnome projects. In *MSR '10: 7th IEEE Working Conference on Mining Software Repositories*, pages 98–101, 2010.

[18] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 314–321. IEEE Computer Society, 1997.

[19] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Analysis of the linux kernel evolution using code clone coverage. In *MSR '07: Proceedings of the 4th International Workshop on Mining Software Repositories*, page 22. IEEE Computer Society, 2007.

[20] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 329–342. ACM, 2011.

[21] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 120. IEEE Computer Society, 2000.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 284–292. ACM, 2005.

[23] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 315–324. ACM, 2010.

[24] A. Ozment and S. E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 7–7. USENIX Association, 2006.

[25] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 59–71, New York, NY, USA, 2006. ACM.

[26] E. S. Raymond. The cathedral and the bazaar. Sebastopol, CA, USA, 1999. O'Reilly & Associates, Inc.

[27] F. M. Reza. *An introduction to information theory.* McGraw-Hill, New York, 1961.

[28] R. Tairas. Code clones literature, http://students.cis.uab.edu/tairasr/clones/literature/.

[29] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring similarity of large software systems based on source code correspondence. In *Proceedings of 2005 Product Focused Software Process Improvement*, pages 530–544, 2005.

[30] J. H. Zar. Significance Testing of the Spearman Rank Correlation Coefficient. *Journal of the American Statistical Association*, 67(339):578–580, 1972.