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

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.

Though forking provides flexibility in taking an existing project to new directions or providing software under different license restrictions, 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.

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 to identify similar edit content in the patches and determines similar edit operation sequences using N-gram matching. 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 be-
tween contributors [3]. Using the version history of three
BSD projects, we investigate the extent of ported changes
from other projects, the number of developers who are in-
volved 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 fol-
loows:

• 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-porteted 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
other projects: on average 59.52%, 58.85%, and 44.85%
of active developers in FreeBSD, NetBSD, and Open-
BSD 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 respec-
tively. 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 modi-
fied 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 required to port changes from other projects is
not insignificant. A considerable amount of time and devel-
oper effort is spent on repeated work across forked projects.
Currently, the upkeep work of porting changes from other
projects 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 evolu-
tion [29] 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 port-
ing 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 anal-
ysis tool, which finds ported edits with 94% precision
and 84% recall. This tool can serve as a basis for as-
sessing 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-porteted edits, the upkeep work of cross-
system porting is significant and involves a large num-
ber of active developers.

As the decision of forking has long-term consequences, we
plan to further investigate the relationship between port-
ing effort and other software metrics such as dependencies,
coupling, people and organization metrics, etc. Since fork-
ing decisions are often made due to license disagreement or
incompatibility, we aim to investigate the implication of li-
cense terms on porting effort or the information flow among
forked projects by combining our automated porting analy-
sis 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. Sec-
tion 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, sev-
eral 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 dupli-
cation in a software system as opposed to the extent of repet-
tive 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]. Canfora et al. investigated the social interaction flow between OpenBSD and other BSD projects [9]. 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-port changes.

**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 [15]. 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. Padoleau et al. [23] 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 messages, 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].

#### 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 $P_a$ and $P_b$ 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 1: The BSD Product Family |
|------------------------|--------|--------|---------|---|
|                          | KLOC   | releases | authors | years |
| FreeBSD                 | 359 to 4479 | 54      | 405     | 18   |
| (R1.0 - R8.2)           |        |         |         |      |
| NetBSD                  | 859 to 4463 | 14      | 331     | 18   |
| (R1.0 - R5.1)           |        |         |         |      |
| OpenBSD                 | 297 to 2097 | 30      | 264     | 16   |
| (R1.1 - R5.0)           |        |         |         |      |
repertoire compares both the content and edit operations of patches.

<table>
<thead>
<tr>
<th>Token Size</th>
<th>Precision</th>
<th>Recall</th>
<th>f-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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 and 5/1/2009, 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.
Table 3: An example of ported edits found by Repertoire. Ported edits are colored in gray

<table>
<thead>
<tr>
<th>Location</th>
<th>Change Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/sys/dev/mii/rgephy.c; revision 1.20</td>
<td>Add RTL8211C(L) support. Disable advanced link-down power saving in phy reset</td>
</tr>
<tr>
<td>src/sys/dev/mii/rgephy.c; revision 1.23</td>
<td>Support for RTL8211C(L) phy from FreeBSD</td>
</tr>
</tbody>
</table>

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. +     } else if (rsc->mii_revision < 2) {
545. +         rgephy_load_dspcode(sc);
546. +     }
547.     mii_phy_reset(sc);
548.     DELAY(1000);

--- 583,604 ---
583. rgephy_reset(struct mii_softc *sc)
584. {
585.     struct rgephy_softc *rsc;
586. +     +
587.     + uint16_t ssr;
588.     +
589.     + mii_phy_reset(sc);
590.     + DELAY(1000);
591. +     if (rsc->mii_revision < 2) {
592. +         rgephy_load_dspcode(sc);
593. +     }
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. +         } else {
601. +             PHY_WRITE(sc, 0x1F, 0x0000);
602. +             PHY_WRITE(sc, 0x0e, 0x0000);

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Project</th>
<th>Committer</th>
<th>ChangeLog</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP 1999/03/26</td>
<td>NetBSD</td>
<td>bouyer</td>
<td>src/usr.bin/eject/eject.c: Oups, complete braindamage yesterday. DIOCEJECT does the right thing for both disks and CDs, it’s just don’t have to call DIOCEJECT before, unless we’re doing a forced eject: DIOCEJECT will check for device use and unlock the door if allowed.</td>
</tr>
<tr>
<td>2008/07/23</td>
<td>OpenBSD</td>
<td>djm</td>
<td>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@</td>
</tr>
<tr>
<td>FN 2009/01/29</td>
<td>OpenBSD</td>
<td>thib</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 5: Sample ported edits found by Repertoire.

<table>
<thead>
<tr>
<th>Date</th>
<th>Project</th>
<th>Committer</th>
<th>ChangeLog</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 1997/04/23</td>
<td>NetBSD</td>
<td>scottr</td>
<td>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. Kcore dump, from NetBSD.</td>
</tr>
<tr>
<td>1999/04/23</td>
<td>OpenBSD</td>
<td>downsj</td>
<td>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. Cleanup of ksh memory handling from OpenBSD via Stefan Krueger in PR 24962.</td>
</tr>
<tr>
<td>2. 2002/03/01</td>
<td>OpenBSD</td>
<td>espie</td>
<td>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. Introduce IPMI and ASF related code from FreeBSD.</td>
</tr>
<tr>
<td>2004/07/07</td>
<td>NetBSD</td>
<td>mycroft</td>
<td>this is a rather large change to add support for the BCM5709. Add support for the Broadcom BCM5709 and BCM5716 chips</td>
</tr>
<tr>
<td>3. 2006/09/09</td>
<td>FreeBSD</td>
<td>ambroko</td>
<td>src/sys/dev/mii/rgephy.c; revision 1.20</td>
</tr>
<tr>
<td>2010/01/28</td>
<td>NetBSD</td>
<td>msaitoh</td>
<td>this is a rather large change to add support for the BCM5709. Add support for the Broadcom BCM5709 and BCM5716 chips</td>
</tr>
<tr>
<td>4. 2009/07/03</td>
<td>OpenBSD</td>
<td>dlg</td>
<td>src/sys/dev/mii/rgephy.c; revision 1.23</td>
</tr>
<tr>
<td>2010/01/27</td>
<td>NetBSD</td>
<td>sborrill</td>
<td>src/sys/dev/mii/rgephy.c; revision 1.23</td>
</tr>
</tbody>
</table>
Precision: the percentage of ported lines found by Repertoire that are also present in the ground truth, i.e., \( \frac{|E \cap R|}{|E|} \).

Recall: the percentage of the ground truth that is also present in the Repertoire’s results, i.e., \( \frac{|R \cap E|}{|R|} \).

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 1 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 2 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 \texttt{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.

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:

\[
\text{avg. porting rate} = \frac{\sum_{\text{releases}} \text{ported edits}}{\sum_{\text{releases}} \text{total edits}}
\]

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.
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 understand porting rate changes, we apply linear regression to 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.

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, corresponding to 14%, 16%, and 11% porting rates on average in FreeBSD, NetBSD, and OpenBSD.

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

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>c</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>1.89</td>
</tr>
<tr>
<td>NetBSD</td>
<td>-16.03</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>1.42</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>-1.27</td>
</tr>
<tr>
<td>NetBSD</td>
<td>68.19</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>58.85</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>58.85</td>
</tr>
<tr>
<td>NetBSD</td>
<td>11.54</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>14.54</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>CLOC</th>
<th>Ported CLOC</th>
<th>Non-Ported CLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeBSD</td>
<td>4754862</td>
<td>654858</td>
<td>4100004</td>
</tr>
<tr>
<td>Correlation</td>
<td>0.26</td>
<td>0.15</td>
<td>0.25</td>
</tr>
<tr>
<td>p-value</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
</tr>
<tr>
<td>NetBSD</td>
<td>4997338</td>
<td>636006</td>
<td>3461342</td>
</tr>
<tr>
<td>Correlation</td>
<td>0.41</td>
<td>0.36</td>
<td>0.42</td>
</tr>
<tr>
<td>p-value</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>4728360</td>
<td>607810</td>
<td>4220500</td>
</tr>
<tr>
<td>Correlation</td>
<td>0.37</td>
<td>0.32</td>
<td>0.38</td>
</tr>
<tr>
<td>p-value</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
</tr>
</tbody>
</table>

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.

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.

![Graph showing percentage of developers involved in porting changes from different projects per release](image)

### 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

![Graph showing percentage of developers involved in porting changes from different projects per release](image)
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 contributors 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 workload 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:

$$\text{normalized entropy} = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

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 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.

$$A \text{ significant portion of active committers port changes from other projects.}$$

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:

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

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.

$$A \text{ significant portion of active committers port changes from other projects.}$$

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

$$\text{Figure 5: The cumulative distribution of ported changes from other projects vs. patch propagation time.}$$

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.

<table>
<thead>
<tr>
<th></th>
<th>FreeBSD</th>
<th>NetBSD</th>
<th>OpenBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>2.96%</td>
<td>2.33%</td>
<td>5.66%</td>
</tr>
<tr>
<td>MED</td>
<td>5.30%</td>
<td>3.96%</td>
<td>6.63%</td>
</tr>
<tr>
<td>Both</td>
<td>3.69%</td>
<td>3.45%</td>
<td>5.64%</td>
</tr>
<tr>
<td>Total</td>
<td>11.58%</td>
<td>11.35%</td>
<td>18.62%</td>
</tr>
</tbody>
</table>

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 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](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-
analyses by German et al. [11,12].

projects by combining our automated analysis with license porting effort or information flow among a group of related projects. We plan to understand the implication of license terms on port changes across peer projects. In terms of future work, we plan to further study the types of adaptations required to support the BSD community seems to heavily depend on developers forking projects or notifying developers of potential collateral evolution. To guide design of such approaches, we forked projects or noticed similar program transformations to related contexts among FreeBSD, NetBSD, and OpenBSD projects. This work was partly supported by National Science Foundation under the grants CCF-1117902, CCF-1149391, and CCF-1043810 and by a Microsoft SEIF award.

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 in isolation.

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].

7. REFERENCES

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.

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

Acknowledgements

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


