The Dissertation Committee for Baishakhi Ray
certifies that this is the approved version of the following dissertation:

Analysis of Cross-System Porting and Porting Errors in Software Projects

Committee:

Miryung Kim, Supervisor

Christine Julien

Sarfraz Khurshid

Dewayne E. Perry

Suzette Person

Neha Rungta

Analysis of Cross-System Porting and Porting Errors in Software Projects

by

Baishakhi Ray, B.Sc., B.Tech., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2013

Dedicated to my husband Suman.

Acknowledgments

I would like to thank my family, friends, and mentors who helped me through out my graduate studies and made my PhD a fun experience.

I am grateful to my advisor, Miryung Kim, for holding my hand in each step of the PhD program. She taught me how to be critical about my own research—how to evaluate a research problem before diving into it, how to question every step while conducting research, and how to present the results in writing form. She encouraged me to explore new areas and gave me the opportunity to collaborate with others. I really enjoyed all those informal discussions with her that later lead to papers. She is a great mentor, collaborator, and friend.

I greatly appreciate the feedback of my thesis committee that helps a lot to improve this work. Without their encouragement and guidance this thesis would not have materialized. A special thank to Suzette Person and Neha Rungta, who were also my mentors in Google Summer Code 2012, for their constant advice and guidance. I would like to thank Sarfraz Khurshid for his constructive comments on the evaluation of the porting error detection algorithm.

I thank Na Meng for the discussions that inspired the design of the SPA algorithm and for her help in reusing the implementation of Sydit and LASE. Her help played a significant role in the design and implementation of SPA.

I am thankful to Christopher Wiley to collaborate with me in designing and

implementing REPERTOIRE. I also want to thank Jihun Park for gathering the bug history data for FreeBSD, NetBSD, and OpenBSD projects.

I thank RoseAnna Goewey to help me submit the necessary paper work for this thesis. Without her help, I could not have met the requirements.

I thank my lab-mates Ripon Saha, Rui Qui, Lisa Hua, Sungmin Cho, and Na Meng who attended my practice talks and helped me improve my presentation skill with valuable feedback.

A special thank to Souriyo Dishak to proof read the thesis.

I sincerely thank my friend Amrita Panda for being there whenever I needed to talk. I also want to thank Tathagata Ghosh and Swamy Ananthanarayan to sharpen my philosophical acumen by years of endless conversation.

I am very grateful to my family for being there through out this journey. My mother always wanted me to pursue Ph.D. and become a scientist. Without her constant encouragement and guidance I could never achieve this. I learn perseverance from my father. His enthusiasm made this journey a lot easier. I would like to thank my sister for honing my teaching skill from childhood and trusting my creative and critical faculty. I am indebted to rest of my family for their enormous support throughout my graduate life.

Finally, I cannot express my gratitude in words to my husband Suman Jana for sharing this journey with me. *Iron sharpens iron, and one man sharpens another* - summarizes this journey.

Analysis of Cross-System Porting and Porting Errors in Software Projects

Publication No. _____

Baishakhi Ray, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Miryung Kim

Software forking—creating a variant product by copying and modifying an existing project—is often considered an ad hoc, low cost alternative to principled product line development. To maintain forked projects, developers need to manually port existing features or bug-fixes from one project to another. Such manual porting is not only tedious but also error-prone. When the contexts of the ported code vary, developers often have to adapt the ported code to fit its surroundings. Faulty adaptations or inconsistent updates of the ported code could potentially introduce subtle inconsistencies in the codebase.

To build a deeper understanding to cross-system porting and porting related errors, this dissertation investigates: (1) How can we identify ported code from software version histories? (2) What is the overhead of cross-system porting required to maintain forked projects? (3) What is the extent and characteristics of porting errors that occur in practice? and (4) How can we detect and characterize potential porting errors?

As a first step towards assessing the overhead of cross-system porting, we implement REPERTOIRE, a tool to analyze repeated work of cross-system porting across peer projects. REPERTOIRE can detect ported edits between program patches with high accuracy of 94% precision and 84% recall. Using REPERTOIRE, we study the temporal, spatial, and developer dimensions of cross-system porting using 18 years of parallel evolution history of the BSD product family. Our study finds that cross-system porting happens periodically and the porting rate does not necessarily decrease over time. 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.

Analyzing version histories of Linux and FreeBSD, we derive five categories of porting errors, including incorrect control- and data-flow, code redundancy, and inconsistent identifier and token renamings. Leveraging this categorization, we design a static control- and data-dependence analysis technique, SPA, to detect and characterize porting inconsistencies. SPA detects porting inconsistencies with 65% to 73% precision and 90% recall, and identify inconsistency types with 58% to 63% precision and 92% recall on average. In a comparison with two existing error detection tools, SPA outperforms them with 14% to 17% better precision.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

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. 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 or bug fix from one program context to another.

Forking is particularly common in free and open source software projects. The open source community often forks an existing project due to a conflict in vision or personality clash. For instance, the split of FreeBSD and NetBSD from 386BSD, XEmacs from GNU Emacs, and LibreOffice from OpenOffice are well known forks. Proprietary software is also forked to support different customer needs. Some notable proprietary forks include EnterpriseDB (a fork of PostgreSQL), Mac OS X (based on the proprietary Nextstep and the open source FreeBSD), and Cedega and CrossOver (proprietary forks of Wine) [80].

Though forking provides flexibility in taking an existing project in new directions or providing software under different license restrictions [21], forking

has negative implications during software maintenance. As multiple peer projects evolve in parallel, development effort is often duplicated in the sense that developers manually port similar features and bug fixes across forked projects [68]. The amount of repetitive work required to maintain forked projects is yet unknown.

Moreover, when porting a patch from one implementation to another, developers need to adapt the patch to fit the new context. The code in the reference patch usually serves as a template that is pasted into the target implementation, and later adapted [39]. The process of adapting a patch to fit another context can be prone to errors, often resulting in *porting errors* [35, 50]. Porting errors also occur when developers evolve ported code differently [28]. Juergens et al. find that *"nearly every second, unintentional inconsistent changes to clones lead to a fault" [36].*

The goal of this thesis is to build a deeper understanding of the extent and characteristic of cross-system porting and porting errors. Based on this understanding, the dissertation presents tool support to monitor cross-system porting and to detect and diagnose potential porting errors. We believe these tools should help managers and architects to make informed decisions on how to maintain forked projects and assist developers in detecting and diagnosing porting inconsistencies.

1.2 Background

Porting introduces duplicate code (i.e., code clones) in the codebase. While studies exist on code duplication [20, 37, 75], we still do not have a clear understanding of the extent and nature of cross-system porting across forked projects. Existing clone detection tools can only identify similar code fragments in soft-

ware [15,16,34,37,44], but the presence of similar code fragments between projects may not always indicate porting. For example, a fresh forked project contains 100% cloned code from its parent, but this does not necessarily indicate that developers ported patches from one project to another to co-evolve the system. While the ported code always produces clones, the converse is not necessarily true. Also, existing empirical studies [20, 37] of code clones focus only on the extent of code duplication but not on the extent of repetitive work involved in porting features or bug fixes from one project to another. For our study, it is useful to identify ported edits rather than code clones across projects, since they are more reliable indicators of repetitive effort involved in maintaining forked projects.

Some prior research focuses on the cross-system interaction of forked projects, but these projects are limited to only analyzing change logs and email messages, not the actual code. For example, Canfora et al. investigated the social characteristics of contributors who make cross-system bug fixes between FreeBSD and OpenBSD [17]. German et al. studied the copyright implications when code fragments transfer between different peer projects under different licenses but did not measure the extent of repetitive work required to maintain the forked projects [29].

When developers port code from a reference to a target context, they usually expect the ported code to behave similarly. Existing tool support to reason whether a ported code is behaving similarly in a target context is limited. Li et al. and Juergens et al. find inconsistent clones using a lexical clone detection analysis [36, 50]. Jiang et al. and Gabel et al. report clone related bugs by comparing the syntax tree structures for two clones [28, 35]. Such syntactic and lexical analyses are not

sufficient to detect the semantic inconsistencies that arise due to incorrect adaptation or inconsistent updates of ported edits in different contexts.

1.3 Solution

This thesis consists of four parts. First, we measure how much repetitive work takes place across forked projects due to cross-system porting. To understand the longitudinal impact of forking on software maintainability, we implement REPERTOIRE, a cross-system porting analysis tool. Second, using REPERTOIRE, we conduct an in-depth case study of cross-system porting in the BSD product family using 18 years of parallel evolution history. Third, we analyze the extent and characteristics of porting errors from the version histories of Linux and FreeBSD as a first step towards detecting and diagnosing potential porting errors. Fourth, we design SPA, a semantic porting analysis tool. Leveraging the understanding of porting errors, SPA analyzes the impact of ported code in its reference and target contexts respectively using control- and data- dependence analysis. Since these impacted code fragments characterize the semantics of the ported edits, SPA compares the impacted reference and target code to detect and diagnose porting inconsistencies.

1.3.1 REPERTOIRE: A Cross-System Porting Analysis Tool

REPERTOIRE analyzes the extent and characteristics of repeated work required to maintain forked projects in terms of cross-system porting. REPERTOIRE identifies how similarly two programs changed as opposed to simply detecting duplicate code between the two programs. Given two program patches as input,

REPERTOIRE identifies the ported edits—code fragments within patches that have similar content and identical edit operations. REPERTOIRE identifies similar edit content between the patches using CCFinderX [37] and determines similar edit operations using *bi-gram* matching [8]. REPERTOIRE further disambiguates the source vs. target of the ported edit by comparing the commit dates of similarly edited code regions.

To evaluate the accuracy of REPERTOIRE, we manually construct a ground truth of ported edits on a sampled data set. The ground truth either contains code changes whose commit messages indicate cross-system porting activities or manually verified ported edits. The comparison between the REPERTOIRE's results and this ground truth finds that REPERTOIRE identifies ported edits with 94% precision and 84% recall.

Analyzing two sets of program patches taken from two forked projects, REPERTOIRE computes the extent of cross-system porting between the two patches. REPERTOIRE further investigates how long it takes for programmers to port a patch, to which sub-directories porting is mostly localized, and which developers port most. REPERTOIRE displays these temporal, spatial, and developer characteristics of cross-system porting activities using various graphical views. It also provides an interactive interface to browse ported edits. Currently it is fully integrated with the state of the art version control systems such as Git [4] and Mercurial [5]. These analyses are designed to aid managers and architects to make informed decisions about the maintenance of forked software systems.

1.3.2 A Case Study of Cross-System Porting in the BSD Product Family

We conduct an in-depth case study of cross-system porting in the BSD product family using their 18 years of co-evolution history to check whether forking is a sustainable practice and to measure the overhead of cross-system porting in real systems. Using REPERTOIRE, we investigate five research questions. The results are summarized as follows:

- **What is the extent of edits ported from other projects?** Porting consists of a significant portion of the BSD family evolution, corresponding to on average 14%, 16%, and 11% of total changes in each release in FreeBSD, NetBSD, and OpenBSD respectively. Porting happens periodically in the BSD family, and the porting rate does not necessarily decrease over time across all three projects.

- **Are ported changes more defect-prone than non-ported changes?** Files with ported edits are less defect-prone than the files with non-ported edits in all three BSD projects. This implies that developers may selectively port well-tested features and bug fixes from peer projects.

- **How many developers are involved in porting patches from other projects?** In each release, a significant percentage of active developers port changes from peer projects, on average 26.12%, 58.85%, and 44.85% 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.

- **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. 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, indicating that porting is concentrated on a few sub systems.

These results indicate that, in the BSD product family, the work required to port changes across peer projects is significant and is heavily dependent on the efficiency and promptness of few core developers.

1.3.3 An Empirical Study of Porting Errors

As a first step towards automatically detecting and diagnosing porting errors, we study the extent and characteristics of porting errors that occur in practice. In our study, we mine the version histories of Linux and FreeBSD to detect commit messages containing keywords related to *porting errors*, in order to understand the types of porting errors and their fixes. We use Sliwerski et al.'s fix-inducing change identification method [74] to identify the patch that originally introduced the porting error. We then use REPERTOIRE [66] to find the reference patch that served as a template for the error-inducing patch. Through a manual investigation of the

reference patch, the error-inducing patch, and the fix patch, we find that many porting errors result from incorrect adaptations of ported code, including inconsistent identifier renamings, different control- and data-flow contexts in the reference and target implementations, and code redundancy. The extent of porting errors is significant in practice. On average, it takes more than a year for a porting error to be detected and fixed. These results motivate the need for an automatic tool to detect porting errors.

1.3.4 Semantic Porting Analysis and Error Diagnosis

Leveraging the characterization of porting errors, we design and implement SPA, an algorithm to detect and characterize porting inconsistencies. SPA detects semantic inconsistencies that arise due to the interactions between program statements in the ported code and the program statements surrounding the ported code. SPA takes two code patches as input: a reference patch and a target patch, each of which characterizes the syntactic differences between two program versions (Ref_{old} and Ref_{new}), and (Tar_{old} and Tar_{new}), respectively. SPA analyzes the reference and target patches to identify ported code and then uses static control- and data-dependence analyses to identify the impact of the ported code on the reference and target contexts. Finally, SPA compares the impact of the ported code on the reference and target semantics to detect and characterize porting inconsistencies.

To evaluate the accuracy of SPA, we perform an empirical evaluation on four large open-source codebases: FreeBSD, Linux, Eclipse CDT, and Mozilla. We compare its results with two state-of-the-art tools, DejaVu [28] and Jiang et al.'s

clone related bug detection tool [35]. These results show that SPA identifies semantic porting inconsistencies with 65% to 73% precision and 90% recall and identifies inconsistency types with 58% to 63% precision and 92% recall on average. SPA outperforms two related error detection tools by 14% to 17% higher precision.

1.4 Contributions

This thesis makes the following contributions:

1. We propose a novel tool, REPERTOIRE, to detect ported edits from program patches. We evaluate REPERTOIRE based on a ground truth set and show that REPERTOIRE finds ported edits with 94% precision and 84% recall.

2. We perform an in-depth case study of cross-system porting in the BSD product family. Our analysis shows 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.

3. We conduct a comprehensive study of the extent and characteristics of porting errors reported for real-world systems Linux and FreeBSD. We identify categories of common porting errors, such as inconsistent control flow, inconsistent data flow, inconsistent identifier renaming, and code redundancy.

4. We design and implement a novel algorithm, SPA, to identify potential porting errors by detecting inconsistent semantics of ported code between the reference and target contexts. We conduct an empirical evaluation of SPA's abil-

ity to detect and characterize porting inconsistencies using four large open-source codebases.

1.5 Potential Impact

Using REPERTOIRE, managers and architects can monitor co-evolution of forked projects. They can measure the frequency of cross-system porting, learn how much duplicated development effort is taking place among peer projects, how much one project is lagging behind its peers, and whether some modules contain more ported code fragments than others.

Our case study of cross-system porting shows that developers spend a considerable amount of time and effort doing repeated work to maintain forked projects. Such upkeep work of porting changes heavily depends on contributors doing their job on time. These results call for automated tool support for cross-system porting. A tool should notify relevant developers of potential collateral evolution, and should help developers in applying a feature implementation or bug fix to relevant contexts in different projects [52, 53].

Our porting error study motivates for automated support to assist developers in porting code correctly. Our porting analysis tool, SPA, serves this purpose. Using SPA, developers can detect porting inconsistencies at an early phase of the development cycle, and thus reduce porting related errors.

1.6 Organization

The rest of this thesis is organized as follows. We start by visiting the related work in Chapter 2. Then we present the implementation and evaluation of REPERTOIRE in Chapter 3. Next, we discuss our empirical study of cross-system porting across the BSD product family in Chapter 4. Chapter 5 presents the study on porting errors in Linux and FreeBSD. Chapter 6 discusses SPA, a tool to detect and diagnose semantic porting inconsistencies. Finally, we conclude in Chapter 7 with a description of future work.

Chapter 2

Related Work

In this chapter, we discuss existing work related to the thesis. First, in Section 2.1 we survey existing literature on porting and code clone analysis. Next, in Section 2.2 we discuss empirical studies on forked software projects. We then review prior work that detects code cloning related inconsistencies and errors in Section 2.3 and Section 2.4. Finally, in Section 2.5, we discuss some existing program differencing techniques that are relevant to our approach in detecting porting inconsistencies.

2.1 Studies on Porting and Code Duplication

Porting Practices. Developers often port a code patch from one implementation to another in order to introduce similar features or bug fixes. Nguyen et al. show that, to fix recurring bugs, developers port code to *code peers*—code fragments with similar functionality or API usage [57]. Also, when libraries and frameworks evolve their APIs, client applications make similar updates to use the new APIs correctly [10]. Kim et al. find that developers frequently port code from one context to another in order to introduce similar functionalities and structural patterns [39]. Also, typically 10% to 30% of the code is similar in a large codebase [37], which

often require similar updates during software evolution [40].

Studies on Code Duplication. When programmers port code from one context to another, it introduces duplicate code to codebases. Existing studies show that 8.7%, 29% and 22.7% of code is clone in *gcc, JDK* and *Linux* respectively [37]. James et al. find evidences of adopted code in device driver modules between Linux and FreeBSD [20]. Gabel and Su investigate source code uniqueness across 6000 projects, and notice that code fragments are similar up to seven lines [27].

However, porting may not be the only source of code duplication. For example, forked projects share a large portion of common code as they originate from a common ancestor. In fact, around 40% lines of code are similar among FreeBSD, NetBSD, and OpenBSD [81]. Not all of this cloned code came through porting, thus does not account for repetitive effort involved in cross-system porting. To measure repeated work, REPERTOIRE investigates how similar are the program patches applied to forked software systems as opposed to finding cloned code between them. This requires REPERTOIRE to identify similar additions, deletions, and modifications between two program patches by considering an extra dimension of edit operation similarity.

Clone Detection Tools. Ported edits are code fragments imported from a different project through an application of similar patch. Such ported code tends to overlap with duplicated code in the codebase, can be detected by existing clone detection tools based on various similarity metrics. For example, tools like Dup, CCFinderX, and CP-Miner [15, 37, 50] identify code clones based on lexical and syntactic similarity. Deckerd and CloneAST find such duplicated code by matching subtrees of

the Abstract Syntax Tree (AST) generated from the source code [16, 34, 44]. Some other clone detectors depend on isomorphic program dependence graph (PDG) analysis [26, 43, 45]. Kim et al. detect semantic clones between two programs by comparing symbolic values of the program variables along all possible execution paths [38]

However, not all cloned code is ported code. Existing clone detection tools cannot make the distinction. To find ported code, we need to find whether similar code has been modified similarly. In other words, detecting ported code involves assessing similarity along two dimensions: code content and edit operation. Existing clone detection tools do not consider the edit operation similarity.

Clone Evolution Analysis. To detect ported code, we check how similarly program patches change, as discussed in Chapter 3. In contrast, existing clone evolution studies monitor how similarly a pre-identified cloned region changes over time [14, 41]. They are broadly classified into four categories: new, modified, never modified, and deleted [46]. Nevertheless, monitoring clone evolution cannot help us to track similar program modifications, because ported code is not necessarily restricted to cloned code.

Clone Visualization Tool. Several studies on code clone detection techniques provide visualization tools to explore clones in the system. For example, tools such as CCFinder-X [37, 51] show a scatter plot [82] of files that contain clones. Such a scatter plot of clone distribution across files helps users to identify where duplicate code is distributed in the file system. Tools like SOFTGUESS and Tarias [9, 76] provide a visualization aid to explore code clones across different versions of a sin-

gle software system. In contrast, REPERTOIRE displays the extent of cross-system porting between two peer projects and displays different characteristics of cross-system porting along spatial, temporal, and developer dimensions.

2.2 Empirical Studies on Forked Software Projects

To measure the overhead of cross-system porting to maintain forked projects, Chapter 4 presents an in-depth case study of cross-system porting in the BSD product family—one of the most well known, long surviving product families created through software forking. Several studies analyzed the inter-communication of the BSD product family earlier, but they are limited to analyze the emails or commit log messages. 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 [23]. Canfora et al. investigated the social characteristics of contributors who make cross-system bug fixes between FreeBSD and OpenBSD [17]. Instead of code flow, some existing research projects looked at other implications of cross-system interactions in the BSDs. German et al. investigated the issue of copyright violations when code fragments transfer between different BSD systems under different licenses [29]. However, none of these projects looked at the repetitive effort required to maintain forked projects in terms of cross-system porting.

2.3 Extent of Porting Errors

When porting a patch from one implementation to another, developers generally need to adapt the patch to fit the new context. The code in the reference patch often serves as a template that is pasted into the target implementation, and then later adapted [39]. The process of adapting a patch to fit another context can be error-prone, often resulting in *porting errors*.

Chou et al. show that porting is an important source of bugs in operating systems [18]. In 65% of the ported code, at least one identifier is renamed, and in 27% cases at least one statement is inserted, modified, or deleted [50]. When developers forget to make such adaptation correctly porting errors may arise [35]. Porting errors also happen when ported code is evolved differently. Juergens et al. [36] conduct an empirical study on the impact of inconsistent clones in a code base. Their interviews with developers confirm that inconsistencies in the found clones are indeed bugs and report that *"nearly every second, unintentional inconsistent changes to clones lead to a fault."*

To understand the extent and characteristics of porting errors that appear in real systems, Chapter 5 presents a case study of porting errors in Linux and FreeBSD. We find 113 and 182 porting errors by mining 18 years and 3 years of FreeBSD and Linux version histories respectively. Our findings are aligned with previous results—a significant portion of operating system errors come through porting [18]. In order to understand the effort required to fix porting errors, we further investigate (1) How long does it take to fix porting errors? and (2) Are the creator and the resolver of a porting error the same people? Next, by manually in-

vestigating the reference patch, the target patch, and the fix patch related to a porting error, we identify some common categories of porting errors related to inconsistent control flow (Type-A), inconsistent identifier renaming (Type-B), inconsistent data flow (Type-C), and code redundancy (Type-D). Existing studies do not investigate characteristics of porting errors in such details.

2.4 Detection of Porting Errors

Li et al. are pioneers in detecting porting errors. Using CP-Miner, a mining based clone detection tool, they find 28 and 23 new bugs in Linux and FreeBSD respectively, which appeared as developers forgot to rename identifiers consistently after porting [50]. Table 2.1 shows an example of inconsistent identifier renaming in ported code. Variable `prom_phys_total`, highlighted in red, should be updated to variable `prom_prom_taken` to preserve the correct behavior. Jablonski et al. [33] detect similar errors by tracking copy-paste code within an Eclipse IDE and by comparing the corresponding AST representations.

Though the results of these studies are aligned with our findings of Type-B inconsistencies, as discussed in Chapter 6, we observe that such inconsistent renaming is a special case of a more general category of porting inconsistencies—forgetting to adapt identifiers according to the target context (Type-B1 and Type-B2). For instance, while renaming an identifier, developers often forget to rename related identifiers. Suppose that a statement `ata = XPORT_ATA` is ported to `sata = XPORT_ATA`; here developers rename `ata` to `sata`, but forget to update `XPORT_ATA` to `XPORT_SATA`. A simple identifier matching technique cannot discover such er-

Table 2.1: An example of porting errors reported by CP-Miner [50]

.

linux-2.6.6/arch/sparc64/prom/memory.c

```
void __init prom_meminit(void) {

 for(iter=0; iter<num_regs; iter++) {
     prom_phys_total[iter].start_adr =
        prom_reg_memlist[iter].phys_addr;
     prom_phys_total[iter].num_bytes =
             prom_reg_memlist[iter].reg_size;
     prom_phys_total[iter].theres_more =
              &prom_phys_total[iter+1];
   }
  ...
  for(iter=0; iter<num_regs; iter++) {
      prom_prom_taken[iter].start_adr =
            prom_reg_memlist[iter].phys_addr;
      prom_prom_taken[iter].num_bytes =
            prom_reg_memlist[iter].reg_size;
      prom_prom_taken[iter].theres_more =
            &Sprom_phys_total[iter+1];
   }
}
```

Variable `prom_phys_total` should be updated to variable `prom_prom_taken` to preserve the correct behavior. The error is highlighted in red.

rors. SPA detects a broader scope of inconsistent renamings by tokenizing function names, file names, and identifier names using a camel case naming convention and mapping corresponding tokens. Our algorithm detects an inconsistency when a token in one context maps to multiple tokens in the other context. For example, when code is ported from `Export.java` to `Import.java`, SPA checks whether all names related to `export` are updated to `import`.

Similar to SPA, Jiang et al. show that an inconsistent context can also cause porting errors [35]. However, their definition of context is limited to the *innermost* control flow construct surrounding the cloned code. They identify syntactic clones using AST level similarity [34], and then detect inconsistencies by comparing the contexts. While their diagnosis partially overlaps with our categorization of porting errors (Type-A and Type-B1), they do not report renaming errors on groups of identifiers (Type-B2), data flow inconsistencies (Type-C), or redundant operations (Type-D). Also, their error detection analysis is purely syntactic and thus suffers from a higher rate of false positives than our semantic control- and data-flow based approach. Our evaluation in Chapter 6 shows that SPA reports 17% better precision and 3% more recall in detecting porting inconsistencies than Jiang et al. on the Eclipse CDT data set.

DejaVu extends the work by Jiang et al. by using several filtering heuristics, such as assessing textual similarity and pruning non-cloned contexts, to improve its precision [28]. As shown in our evaluation, SPA's error detection still outperforms DejaVu with 14% better precision. Also, DejaVu does not report potential error types while SPA characterizes the detected inconsistencies to help developers reason

about porting errors.

Using a machine learning technique, Wang et al. predict whether it is safe to port code to a target location [77]. Studying clone evolution across different software versions, they first build a code clone genealogy, similar to Kim et al. [41]. A clone family is considered to be *harmful* if its members have changed consistently because the newly ported code is likely to follow the same trend. In contrast, a clone family where clones have either changed inconsistently or not changed at all is considered harmless. Learning clone evolution patterns from the history, they predict *harmfulness* of the ported code using the Bayesian network, a machine learning based approach. This model only determines the risk factor of ported code, but cannot detect actual porting errors. Also, this model is suitable for inconsistent evolution of cloned code but not applicable to adaptation mistakes.

2.5 Program Differencing Tools

Existing program differencing tools are used to identify differences between two program versions [12, 24, 31, 64, 83]. The impact of such program differences to other parts of the source code can be determined by change impact analysis tools using dependency analysis [12, 13, 48, 60, 69]. Semantic program differencing tools analyze source code changes and their impact in terms of execution behavior. For example, SymDiff checks input-output equivalence of two procedures, i.e., given identical input parameters and global values, SymDiff verifies whether the return values and resulting global values are identical. If not, it reports a set of intraprocedural paths that cause violations [47]. Ramos et al. [65] check equivalence be-

tween two programs using a symbolic execution technique and generate test cases showing differential behavior. Using symbolic execution, Person et al. characterize program behavior as constraints on program inputs and resulting effects on program states [62]. The semantic differences between two program versions are then determined by the path-effect constraints, which are present in one version but not in the other. To optimize this approach, DiSE first identifies program statements within a method that are impacted by code change using a static control and data dependence analysis. DiSE then drives symbolic execution on the impacted set to generate path-effect constraints [63]. The resulting constraints characterize semantic differences between two program versions. iDiSE extends DiSE to compute semantic difference beyond method boundaries [73].

However, these tools cannot answer whether two programs have changed similarly. SPA uses both program differencing and change impact analysis as underlying means to detect porting inconsistencies (See Chapter 6). Given two versions of a program, SPA uses ChangeDistiller, an AST based program differencing tool, to identify edited AST nodes [24]. Next, it compares the AST types and labels of the reference and the target edits to identify ported AST nodes. Then using static control and data dependence analysis with respect to the ported nodes, SPA identifies the impacted node set that may characterize the semantics of the ported nodes. Finally, SPA compares the reference and target semantics to compute porting inconsistencies. Chapter 6 also discusses the possibility of extending SPA by adapting DiSE such that differential behavior of ported edits in reference and target contexts can be illustrated by test cases.

Chapter 3

REPERTOIRE: A Cross-System Porting Analysis Tool

Software forking occurs when a developer or a group of developers splits off software into separate conceptual entities by copying and modifying an existing project. To maintain the forked projects, developers often port similar code or bug-fixes from one project to another. This chapter presents REPERTOIRE, a tool that analyzes the extent and characteristics of cross-system porting. REPERTOIRE can also be used beyond identifying cross-system porting because REPERTOIRE can detect ported edits between any arbitrary program patches. For example, we used REPERTOIRE to analyze supplementary bug fix—bugs that are fixed more than once and investigate whether the supplementary fixes have content similar to initial fixes. Section 3.6 discusses this work.

3.1 Problem Statement

To create a large software system, a group of developers usually contributes incremental changes to the latest version of the system. At any moment, this process of software evolution may be interrupted by the creation of a *fork* in the software. Forking occurs when developers create a new project by copying an existing one.

After the fork, the two projects continue to evolve in parallel, and the internal assumptions and functionality may diverge into two very different systems with only a passing resemblance to the shared ancestor.

Forking is particularly common in free software projects, where differing visions of the project directions and personality clashes occur without a unifying profit motive. For instance, the split of 386BSD into FreeBSD, NetBSD, and OpenBSD, the split of XEmacs from GNU Emacs, and the split of LibreOffice from OpenOffice.org are all important forks in open source history. Software developed by industry may also be forked to support the needs of multiple customers with different feature requirements. Other forks, such as the MariaDB fork from MySQL, and LibreOffice from OpenOffice.org occur because of uneasiness with corporate influence [80].

Forks are generally thought of as being *bad* for software projects, because they reduce the total pool of developers available to any one software system, slowing evolution [68]. To maintain co-evolution, developers need to monitor the peer projects and when a necessary feature or a bug fix is introduced to one project, developers may need to port it to the other. This involves lot of manual repetitive work. However, it is not entirely clear what is the overhead involved in cross-system porting to maintain forked projects in parallel.

This work presents REPERTOIRE, a tool which facilitates the analysis of software forks using several quantitative measures. REPERTOIRE allows users to analyze the number of lines of code ported between forks, the developers responsible for those ports, and the time between when a patch is first generated for an

intended project and when it is ported. It also presents a graphical means for users to observe a pattern of ports. These analyses are designed to aid managers and architects to make informed decisions about the extent of porting within projects.

To perform such cross-system porting analysis, first REPERTOIRE identifies ported code in the forked projects. Though ported code introduces duplicate code in software, not all duplicate code necessarily means ported code. Ported code fragments are a subset of cloned code fragments that came from another project through application of similar patches. Existing clone detection tools cannot distinguish ported code from the cloned code [15, 20, 37, 50].

To identify repetitive effort, REPERTOIRE needs to detect how much code has changed similarly between the project peers, not just the amount of similar code in them. Hence, REPERTOIRE takes program patches as input, in contrast to programs. Here, by program patches we mean the difference between two versions of a program, generated by the widely used *diff* tool. REPERTOIRE then detects the similarly edited lines among the input patches in two phases. First, REPERTOIRE uses CCFinderX [37], a widely known clone detection engine, to determine the similar edit content. In the second phase, REPERTOIRE compares the edit operations of the cloned code fragments to identify similar edits. Finally, REPERTOIRE compares the commit dates of the similar edits to disambiguate the source and target and infer ported edits. By associating different metadata information such as developer, time, and location of porting, REPERTOIRE further investigates the extent of ported edits between two projects, how long it takes to port patch, which developers port most, and which subsystem contains most ported code.

This chapter presents REPERTOIRE in details. Section 3.2 describes the methodology to identify ported edits from program patches and Section 3.3 evaluates REPERTOIRE's accuracy based on some manually created ground-truth set of ported edits. Section 3.4 describes REPERTOIRE's features and the sorts of questions users may answer with the tool. Section 3.5 discusses how REPERTOIRE gathers and analyzes data for the visualizations outlined in Section 3.4. Section 3.6 presents other application areas where REPERTOIRE can be useful. Finally, Section 3.7 summarizes REPERTOIRE.

3.2 Methodology

To detect ported edits between two program patches, REPERTOIRE determines similar edit content and operations between them. By program patches, we mean *diff*-based, line-level differences per file. Consider two input patches P_x and P_y shown in Table 3.1. REPERTOIRE identifies similar program modifications between P_x and P_y in the following three steps.

1. **Identify cloned regions between patches.** Since CCFinderX can detect cloned code in C, C++, Java, and Cobol programs, REPERTOIRE first pre-processes *diff*-based patches to convert them into a CCFinderX compatible format. It removes symbols 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 [37] on the pre-processed patches, REPERTOIRE

Table 3.1: REPERTOIRE compares both the content and edit operations of patches.

P_x	P_y
**** Old ****	**** Old ****
X1 for(i=0;i<MAX;i++){	Y1 for(j=0;j<MAX;j++) {
X2 ! x = array[i] + x;	Y2 q = p + q;
X3 ! y = foo(x);	Y3 ! q = array[j] + p;
X4 - x = x - y;	Y4 ! p = foo1(q);
X5 }	Y5 }
**** New ****	**** New ****
X6 for(i=0;i<MAX;i++) {	Y6 for(j=0;j<MAX;j++) {
X7 + y = x + y;	Y7 q = p + q;
X8 ! **x = array[i] + x;**	Y8 ! **q = array[j] + q;**
X9 ! **y = foo(x,y);**	Y9 ! **p = foo1(p,q);**
X10 }	Y10 }

REPERTOIRE Steps

① Identify cloned regions between patches
(lines X2 to X3, lines Y3 to Y4)
(lines X6 to X10, lines Y6 to Y10)
~~(lines X7 to X8, lines Y2 to Y3)~~

② Retrieve edit operation sequences from the cloned regions.
(lines X2 to X3; edit operations: `'!!'`)
(lines Y3 to Y4; edit operations: `'!!'`)
(lines X6 to X10; edit operations: `' +!! '`)
(lines Y6 to Y10; edit operations: `' !! '`)

③ Identify similar edit operation sequences using the bi-gram matching
(lines X2 to X3, lines Y3 to Y4) : similar deletion
(lines X8 to X9, lines Y8 to Y9) : similar addition

Similarly added lines are marked in **red** and similarly deleted lines are marked in blue.

finds a set of cloned region pairs across the input patches. Given the two patches, P_x and P_y in Table 3.1, 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).

2. **Retrieve edit operation sequences from the cloned regions.** For the identified cloned regions, REPERTOIRE identifies edit operation sequences, containing symbols `+`, `-`, `!`, and `n` (signifies unchanged lines). 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. Lines X2 to X3 produce a sequence of two modifications, noted as `'!!'`. Lines X6 to X10 produce a sequence of three edit operations, noted as `'n+!!n'` because X6 and X10 are unchanged lines. Lines Y3 to Y4 produce a sequence of two modifications, noted as `'!!'`. Lines Y6 to Y10 produce a sequence of two modifications, noted as `'nn!!n'` 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 [8] that detects the common bi-grams between two strings. We use a bi-gram matching instead of the longest common subsequence algorithm [32], 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 3.2 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, 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. Table 3.3 shows some examples of ported edits found by 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 data set. We choose an 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 evolution history of 11/1/2008 to 5/1/2009, which corresponds to 4.4 and 4.5 release dates, we search for keywords 'NetBSD'

Table 3.2: 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 3.3: Examples of ported changes found by REPERTOIRE.

Date	Project	Committer	ChangeLog
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**.
2002/07/26	NetBSD	onoe	Add support of Silicon Image 0680 Ultra ATA/133 ATA Controller. It's ugly that all register values are written in numeric, but I can't find any definition of the registers to be written in literal.
2002/09/09	OpenBSD	gluk	Add support of Silicon Image 0680 Ultra ATA/133 Controller. Code from **NetBSD**.
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.
2008/07/02	FreeBSD	yongari	Add RTL8211C(L) support. Disable advanced link-down power saving in phy reset
2009/01/09	NetBSD	cegger	Support for RTL8211C(L) phy. From FreeBSD. [cegger 20090109]
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.**
2001/11/09	NetBSD	augustss	Fix a bug in xfer abort processing when the HC executes ahead of what the driver aborts. Don't block RHSC interrupts.
2002/04/07	FreeBSD	joe	MF**NetBSD**: revision 1.106 date: 2001/11/09 15:01:57; author: augustss; state: Exp; lines: +73 -57 Fix a bug in xfer abort processing when the HC executes ahead of what the driver aborts Don't block RHSC interrupts.
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

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. The token threshold determines the minimum length of the detected code clones. 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 remove 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.

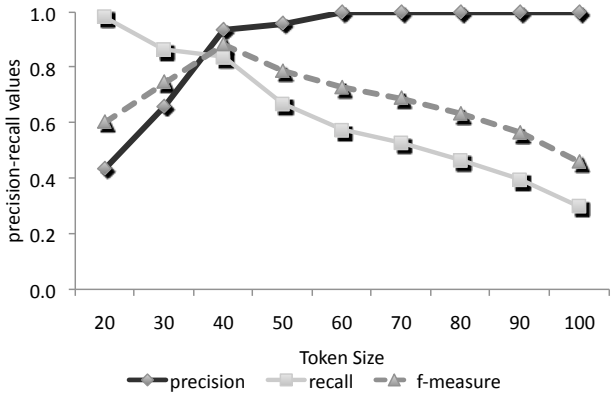


Figure 3.1: Precision and recall values of ported edits found by REPERTOIRE while varying the token threshold values from 20 to 100 tokens

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

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 F-measure (the harmonic mean of precision and recall) reaches a maximum value of 0.88. Hence, we use this threshold of 40 tokens throughout the empirical study in Chapter 4 for optimal accuracy. At token size 40, the precision value is 0.94 and the recall value is 0.84.

Table 3.4 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 false positives, REPERTOIRE detects ported edits between the two patches, though there is no semantic similarity between surrounding contexts. Such a false positive was found because false positive clones could be found by CCFinderX. In the case of false negatives, REPERTOIRE was not able to detect ported edits, because the contiguous lines of ported edits are less than 40 tokens (approximately 10 lines) long.

Table 3.4: Examples of a false positive and a false negative reported by REPERTOIRE

	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**.

3.4 Repertoire Features

To analyze the overhead of cross-system porting, REPERTOIRE associates the identified ported edits with developers, commit dates, and file locations, and characterizes cross-system porting in developers, temporal, and spatial dimensions. This helps project managers or architects to make an informed decision on how to maintain a product family. This section explains how REPERTOIRE can help managers monitor cross-system porting with an example scenario.

Suppose Sheryl is a manager working for Exemplar corporation, which writes and sells software to enterprise customers. Two years ago, a particularly large customer requested a feature that required extensive modifications to the main product. To accommodate this customer's needs, the company forked the main product

and made the necessary custom changes. Since then, a considerable amount of engineering effort has been continually spent to port bug fixes and security patches from the main product. Sheryl is contemplating whether it would be worthwhile to merge the two products back instead duplicating maintenance effort.

Sheryl may need to analyze how the products evolve in parallel and how often cross-system porting occurs. She needs to know where the porting effort is focused, who are the main developers porting code from peer projects, and how often cross-system porting happens. She may also be interested in knowing how long it takes for bug fixes and security patches to propagate from one product to the other. These are the questions that REPERTOIRE can help Sheryl to answer. In the rest of the section, we refer to the main project and the forked project as A and B respectively.

3.4.1 Porting Frequency View

Suppose that Sheryl wants to know how much porting work is actually going on. Given the version histories of two projects, A and B, REPERTOIRE provides a view to visualize the extent of code ported from one project to another over the available history as shown in Figure 3.2. This is represented as a line diagram where x-axis shows time in month and the y-axis shows the average percentage of ported edits with respect to total edits in each commit.

Using this view Sheryl can see how much of the code committed in a month is original and how much is ported. Sheryl may select to see only the edits ported from A to B, only the edits ported from B to A, or both ways. She may see that 90%

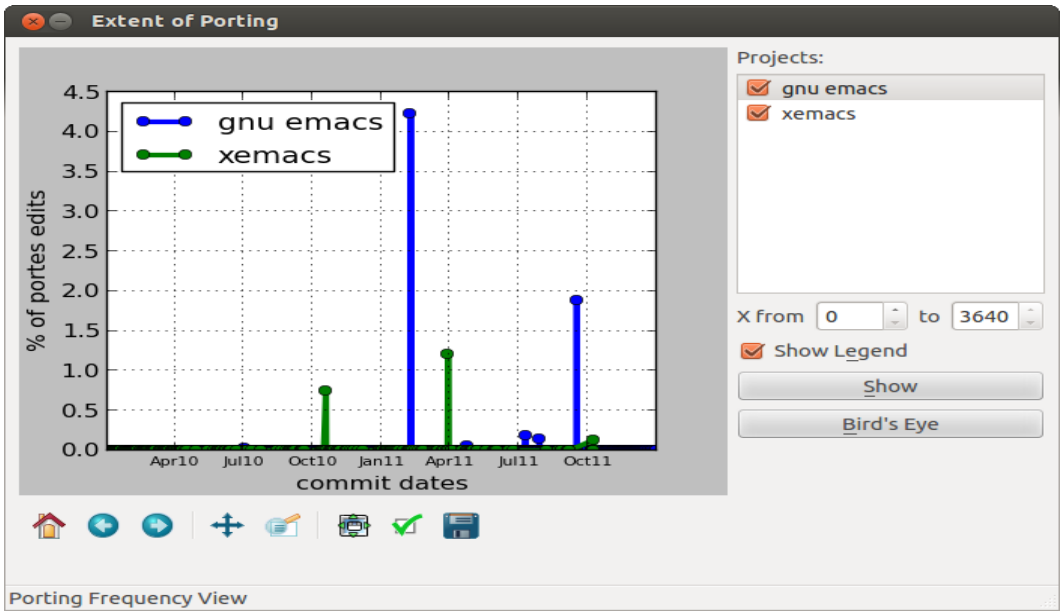


Figure 3.2: Porting frequency view. The X-axis is a time line and the Y-axis is the percentage of edited lines in patches ported from other projects.

of the commits on B are ports, whereas 95% of the commits on A are original code, indicating that most of the engineers working on B spend all their time porting code and very little time writing original code. On the other hand, if Sheryl notices that most edits to either system are not ported, then she may conclude that the systems are diverging further apart.

Figure 3.2 represents the frequency of ported edits between XEmacs and GNU Emacs from February 2010 to December 2011. The results indicate that cross-system porting is not significant between the two projects. The maximum porting took place in March 2011 when only 4.25% of total edits were ported from XEmacs to GNU Emacs. Similarly, in September 2011, only 1.98% of total edits were ported from GNU Emacs to XEmacs.

3.4.2 File Distribution View

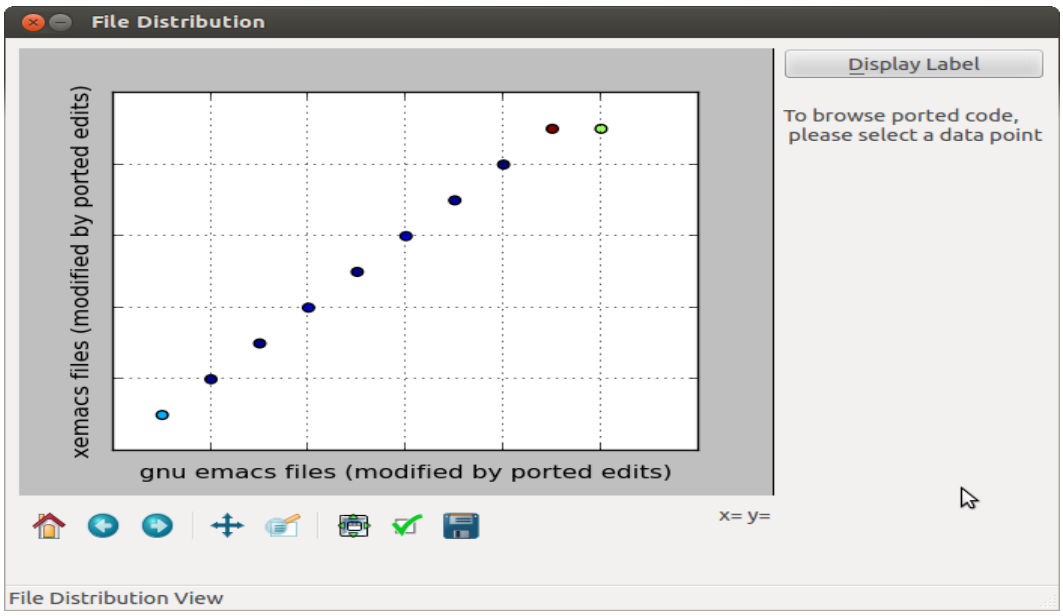


Figure 3.3: File distribution view. A point is plotted for each pair of files with ported edits between them.

To figure out where her organization is spending time porting code, Sheryl needs to see which pairs of files share ported edits between A and B. REPERTOIRE helps Sheryl by showing her a *file distribution* of porting work. This view is a scatter plot with files from project A making up the x-axis and files from project B making up the y-axis. A point is plotted at (x,y) if there is a port from file x to file y or vice versa. We provide users the ability to choose to see only edits going from A to B and vice versa. Users can also see a weighted point where the weight of the point is indicated by the color of the dot. The weight represents the ratio of ports to total edits. The darker the color is, the higher the ratio of ported edits to the total lines of code in the file. Figure 3.3 shows an example of this view of porting.

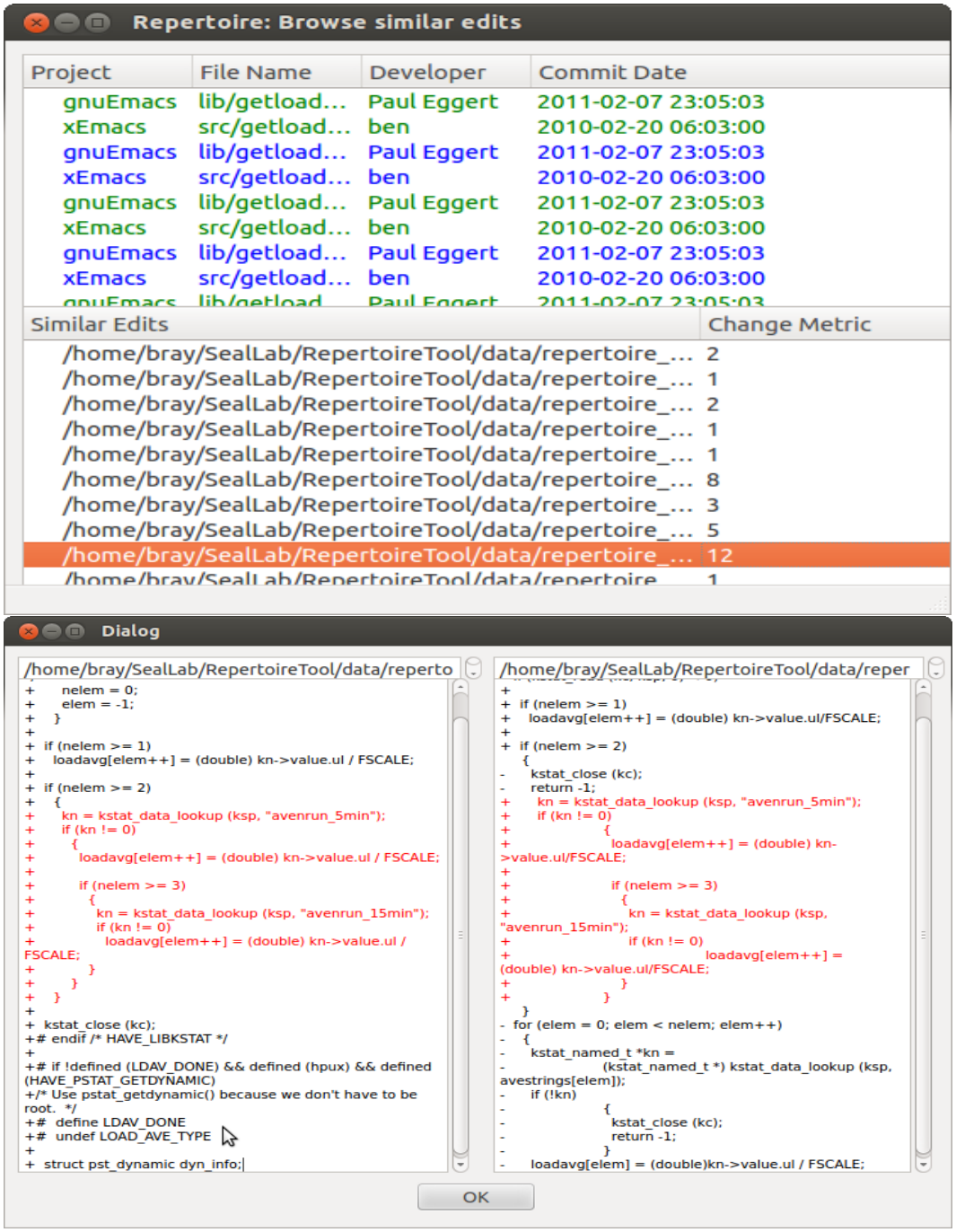


Figure 3.4: The number of ported lines, the corresponding developer, and the dates of the original patch and the ported patch are shown.

By selecting any point on the file distribution view, Sheryl can browse all ported lines between the two files and investigate who ported the corresponding code and when. For example, Figure 3.4 lists all the patches ported from XEmacs file `src/getload.c` to GNU Emacs file `lib/getload.c`. The first two rows show that a code fragment was originally introduced to XEmacs on `2010/02/20` by the developer `ben`, and was ported to GNU Emacs by `Paul Eggert` on `2011/02/07`. The corresponding patches are presented at the bottom part of the figure. The ported edits are highlighted in red. The combination of this view and the file distribution view gives Sheryl a reasonable idea of which code has been ported between projects.

3.4.3 Porting Developer View

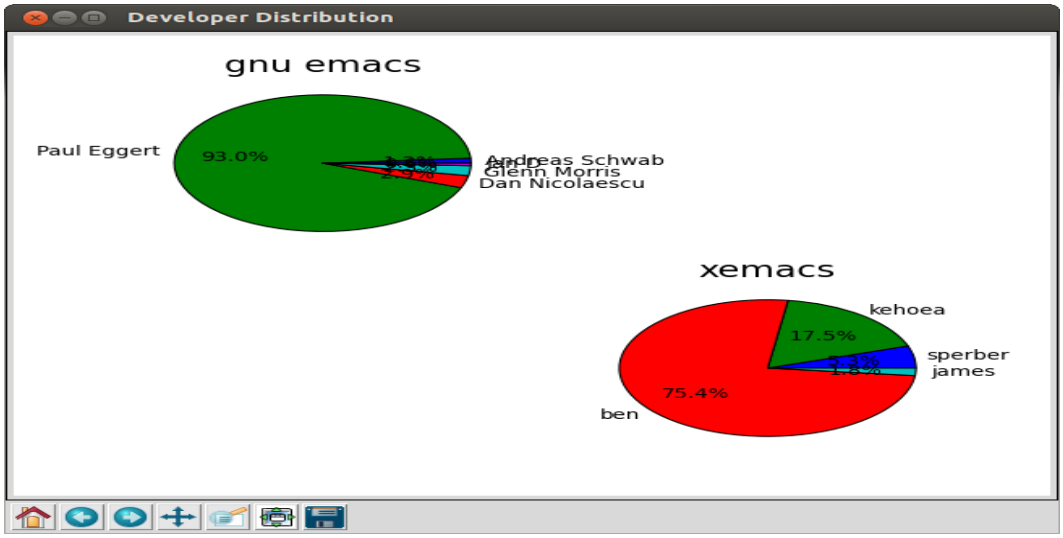


Figure 3.5: Developer view showing which developers are responsible for what fraction of ported edits.

To ask her team about the feasibility of merging A and B, Sheryl may need to identify the developers who have a deep understanding of both projects. A reasonable heuristic for finding such developers is to simply identify the developers doing a lot of porting work. REPERTOIRE displays a pie chart showing which developers are responsible for what fraction of the total ported lines. Figure 3.5 presents developer distribution view for XEmacs and GNU Emacs. It shows that developers `Paul Eggert` and `ben` have done most of the porting work—93% and 75% of the total porting work in GNU Emacs and XEmacs respectively.

3.4.4 Porting Latency View

REPERTOIRE shows a user how long it takes for individual patches to propagate from one system to another system by presenting a cumulative distribution of porting latency. The Porting Latency View shows the number of days between when a patch is first committed to one system and when a similar patch is committed to a target system. Using this view, if Sheryl notices that patches usually take 3 months to propagate, then she knows that users of the one project may experience 3 months of un-updated application behavior than the users of the other.

Figure 3.6 presents the cumulative latency of porting for GNU Emacs and XEmacs. It shows that GNU Emacs took 225 days to port 40% of the total ported edits from XEmacs. However, to propagate all the ported code, it took around 600 days for GNU Emacs. In case of XEmacs, all the ported edits were propagated from GNU Emacs within 200 days.

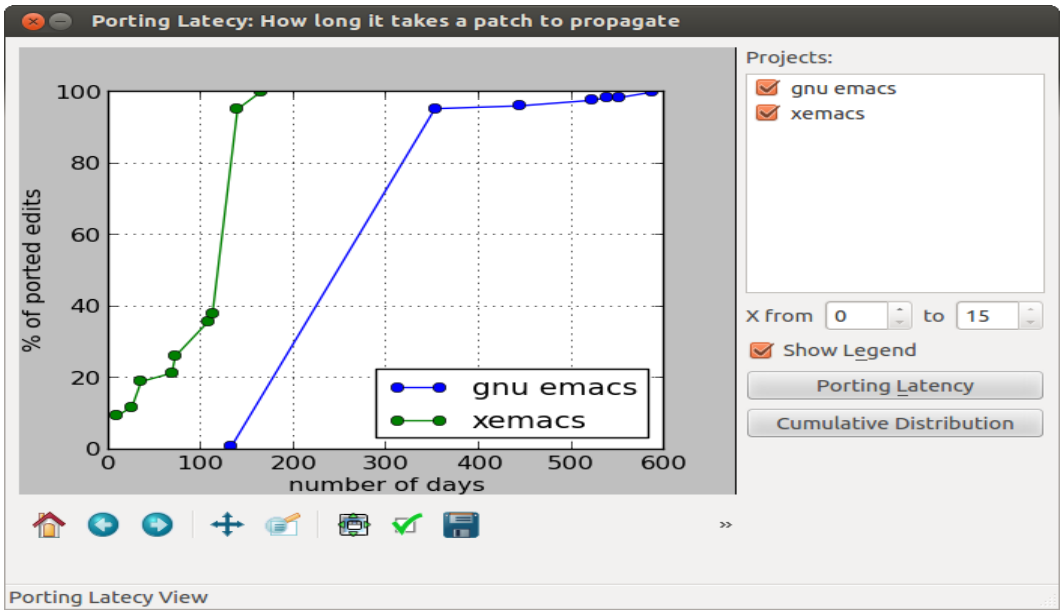


Figure 3.6: Porting latency view showing the cumulative distribution of the time taken to port a patch. A porting latency is the time between the commit of an original patch and the commit of a ported patch.

3.5 Implementation Details

REPERTOIRE runs on Linux, is written in Python, and uses the Qt framework to present a graphical interface to the user. The application is cleanly divided into user interface and back-end layers. The user interface comprises of an *Input Wizard* and an *Analysis Wizard*. The input wizard gathers information about the version histories from the user. The analysis wizard allows the user to perform analysis on the subset of version history she may be interested in. The back-end takes an object with the parameters given by the user and extracts version histories and other metadata from the version control repository. After data processing is complete, the back-end outputs a database which can be loaded from the analysis portion of the UI. This structure is shown in Figure 3.7. The structure of this database is optimized to make queries allowed by the analysis wizard to be run in $O(N)$, where N is the number of ports found. We briefly summarize the working of the back-end below.

3.5.1 Parameters

REPERTOIRE takes a number of parameters as input. The tool needs to have a working directory to store intermediate files, a path to a CCFinderX executable, and information about version control repositories of the two projects under analysis. For each repository, the user is asked to specify what kind of repository it is, where the root of the repository is, and the time window of version history the user is interested in. Table 3.5 shows example inputs to the data analysis stage.

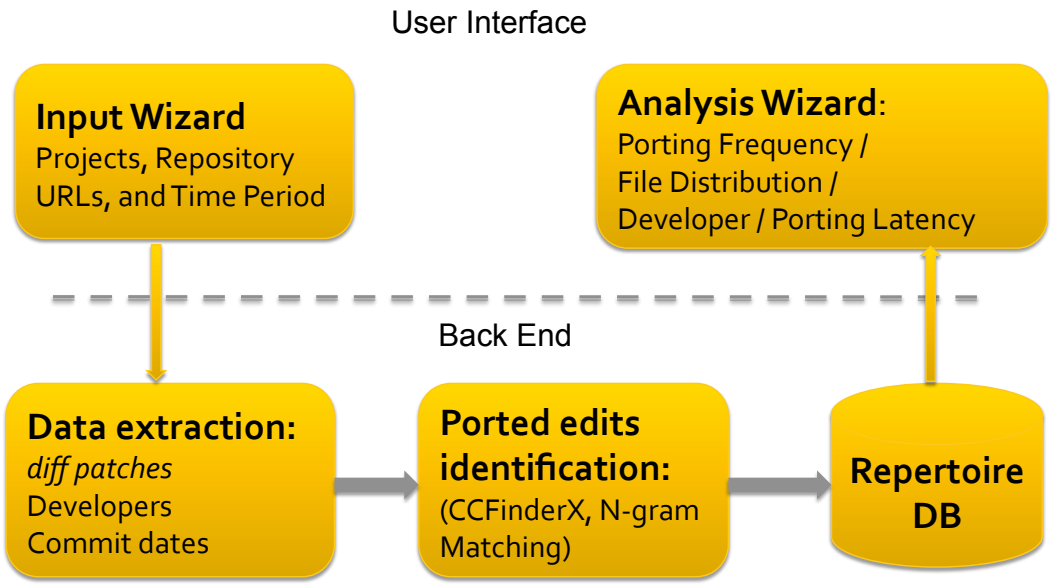


Figure 3.7: REPERTOIRE internal component.

3.5.2 REPERTOIRE Backend

Given the above parameters, REPERTOIRE identifies ported edits in the version histories of the two projects. For each repository, it first extracts the author, date, and change set of each commit in the specified time window. Change sets are simple line level differences generated by the *diff* tool. The change sets are run through REPERTOIRE port identification phase, which identifies similar edits between the two input projects. The port identification process is explained in Section 3.2. After identifying ported lines in the changesets, REPERTOIRE associates them with the author and date metadata taken from the version control system earlier, and puts all these information into a database, called `Repertoire DB`. The database is designed to facilitate quick access by the analysis UI.

	Input Types	Example Inputs
	working directory	`/var/tmp`
	CCFinderX path	`/usr/bin/ccfx`
Project 1	version control type	Git
	repository root	`/path/to/myrepo`
	time period	11/2/2010 - 9/31/2011
Project 2	version control type	Mercurial
	repository root	`/path/to/anotherrepo`
	time period	11/2/2010 - 9/31/2011

Table 3.5: Example inputs to REPERTOIRE.

The database is a simple Python object serialized to a file using the canonical `pickle` module. [1] The basic schema of the database is shown in Figure 3.8.

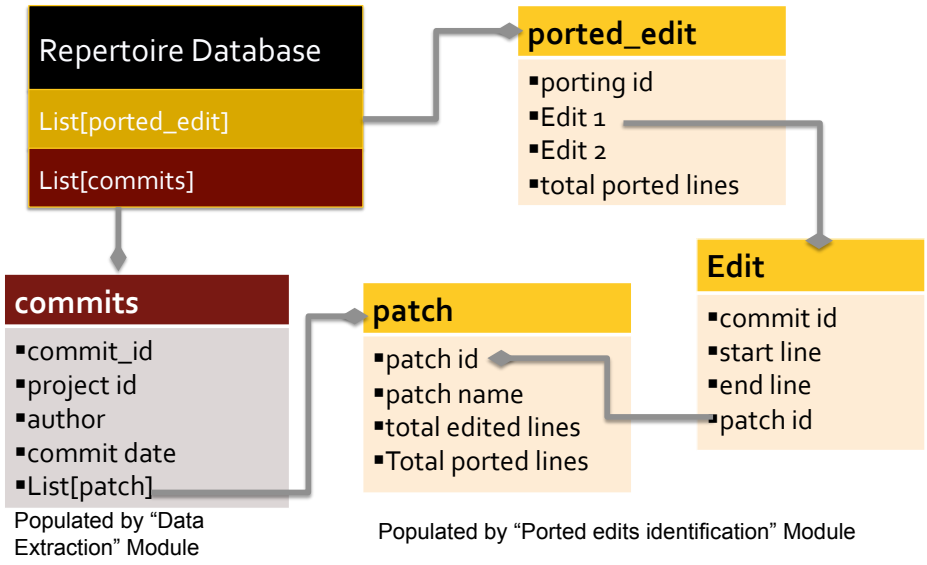


Figure 3.8: REPERTOIRE database schema

[1] http://docs.python.org/library/pickle.html

3.5.3 REPERTOIRE Frontend: User Interface

REPERTOIRE front-end takes the `Repertoire DB` as input and answers different user queries. Figure 3.9 shows the front-end. The GUI provides four analysis views: Porting Frequency View, File Distribution View, Developer View, and Porting Latency View. The details of each view is discussed in Section 3.4.

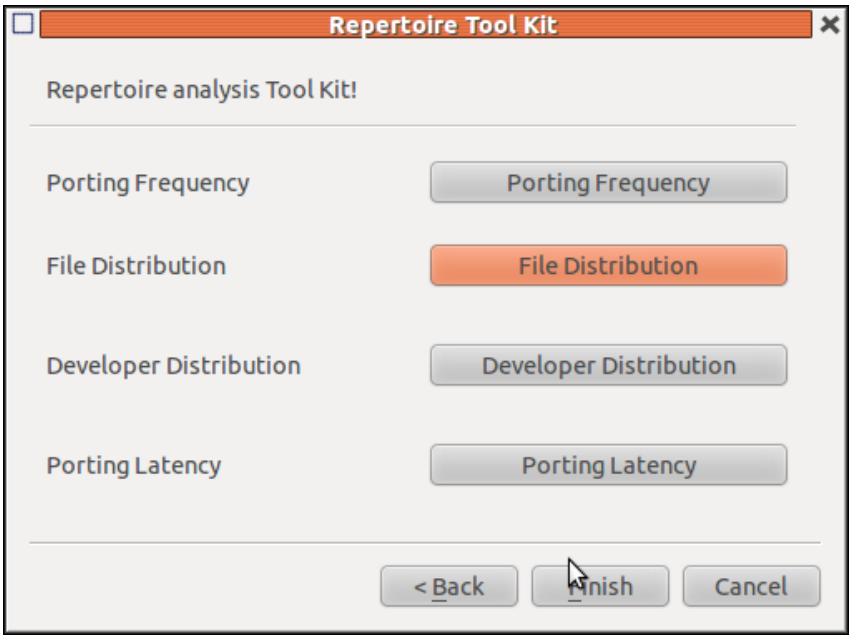


Figure 3.9: REPERTOIRE analysis menu

REPERTOIRE is an open source tool and can be downloaded from `http://dolphin.ece.utexas.edu/Repertoire.html`. A detailed description on how to run the tool is described in Appendix A.

3.6 Other Applications: An Analysis of Supplementary Patches

In practice, REPERTOIRE is not limited to analyzing cross-system porting only. REPERTOIRE can find similar edits within a set of program patches. For example, we used REPERTOIRE to analyze the characteristics of supplementary bug fixes—bugs that are fixed more than once [61].

In general, it is difficult to conclude whether a code change is complete. Developers often omit necessary changes while fixing a bug, resulting in to supplementary patches later [25]. Our study shows that a significant portion of the resolved bugs require supplementary patches (22% in Eclipse JDT core, 24% in Eclipse SWT, and 33% in Mozilla). We performed an in-depth case study to investigate the characteristics of such incomplete fixes.

It is commonly believed that cloned code is one of the primary sources of such *omission errors* [36, 49]. When developers port edits from one context to another, clones are introduced in the system. Later maintaining these clones becomes difficult and error-prone: when one of the similar code fragments is updated, developers sometimes forget to update all the related clones. Thus, a bug-fix that modifies one clone fragment has to be reopened, in order to make similar changes to the other cloned region.

In order to verify whether duplicated code is the primary reason of supplementary bug fixes, we investigate whether a supplementary patch is similar to its original patch. We first identify the bugs that are fixed more than once using the version histories of Eclipse JDT core, Eclipse SWT, and Mozilla. We then use

REPERTOIRE to find similar edits between original bug-fix patches and their supplementary patches. We set the token threshold of REPERTOIRE such that similar patches have at least five similar edited lines.

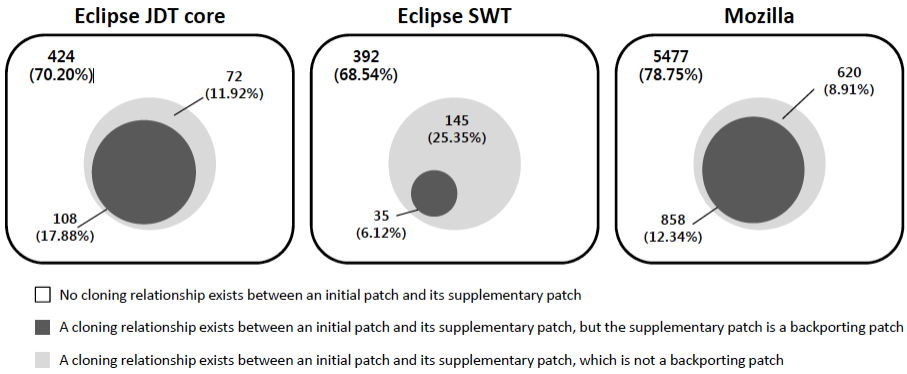


Figure 3.10: The percentage of cloned patches and backported patches out of all supplementary patches, identified by REPERTOIRE. The figure is taken from Park et al. [61]

Figure 3.10 shows the results. 11.92%, 25.35%, and 8.91% of supplementary patches at least have five lines of similar edits with their initial patches, in Eclipse JDT core, Eclipse SWT, and Mozilla respectively. REPERTOIRE also identifies backporting patches—patches identical to initial patches, but applied to different branches. Backporting patches include 17.88%, 6.12%, and 12.34% of the supplementary patches respectively. From these results, we conclude the majority of supplementary patches, 70.20%, 68.54%, and 78.75%, are not similar to their initial patches.

These results contradict the conventional wisdom that code cloning is the primary source of omission errors and supplementary change locations can be pre-

dicted alone by clone-management tools [22,57,58]. In fact, our study finds that the reason of omission errors are diverse including missed porting changes, incorrect handling of conditional statements, and incomplete refactorings.

REPERTOIRE was used to analyze the content similarity between supplementary patches and their initial fix attempts. This shows the utility of REPERTOIRE beyond analyzing ported edits among forked software projects.

3.7 Summary

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.

REPERTOIRE computes the amount of edits that are ported from other projects as opposed to the amount of code duplication across projects.REPERTOIRE takes *diff*-based program patches and using CCFinderX [37] it identifies similar edit content in the patches. It then determines similar edit operation sequences using *bi-gram* matching [8]. 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 examine 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, managers and engineers can measure the frequency of cross-system porting, learn which developers do how much of the porting work, investigate the trend of cross-system porting work over time, and view the spatial distribution of ported edits with respect to the file system structure. REPERTOIRE serves as an end-to-end cross-system porting visualization tool, which analyzes repetitive efforts involved in cross-system porting in developers, temporal, and spatial dimension.

Chapter 4

A Case Study of Cross-System Porting in the BSD Product Family

Though forking provides flexibility in taking an existing project to new directions or providing software under different license restrictions [21], 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 [68]. As a first step towards assessing whether forking is a sustainable practice, we investigate the extent and characteristics of repeated work required to maintain forked projects in terms of cross-system porting. Using REPERTOIRE, we conduct an in-depth case study of 18 years of parallel evolution history of FreeBSD, OpenBSD, and NetBSD—one of the most well known, long-surviving product family created through software forking. We investigate the extent of changes ported from other projects, the number of developers who are involved in porting patches, the time taken to port patches, the locations where ported changes are made to, and the correlation between ported code and bug fixes. 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 chapter presents the study results in detail.

4.1 Study Subject

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. FreeBSD, OpenBSD, and NetBSD are perhaps the most famous "software forks" that evolved from the same codebase. Due to differences of opinion over the future direction and release schedule of 386BSD, FreeBSD project was forked in 1993 from 386BSD [3]. Around the same time, the NetBSD project was founded by a different group of 386BSD users, with the aim of unifying 386BSD with other strands of BSD development into one multi-platform system [6]. Both projects continue to this day. Later in 1995, OpenBSD was forked from NetBSD based on some personal conflicts. OpenBSD is believed to support better security features than its other product family [7].

We use 54, 14, and 30 releases from FreeBSD, NetBSD, and OpenBSD and thus covering 18 years of parallel evolution history. Table 4.1 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 the heuristic developed by Mockus and Votta [54].

Table 4.1: The BSD Product Family

	KLOC	releases	developers	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

4.2 Study Result

This section describes the characteristics of ported code changes in the BSD family. Section 4.2.1 describes the extent and frequency of ported changes. Section 4.2.2 compares defect density of ported changes against non-ported changes. Section 4.2.3 describes the work load distribution of developers who port changes from other projects. Section 4.2.4 describes the time taken to port patches from other projects, and Section 4.2.5 describes the code locations where ported changes were made to.

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

To understand the overhead of repeated work involved in cross-system porting, first we investigate the extent of changes ported across the three BSD project. To do that, we compare 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 re-

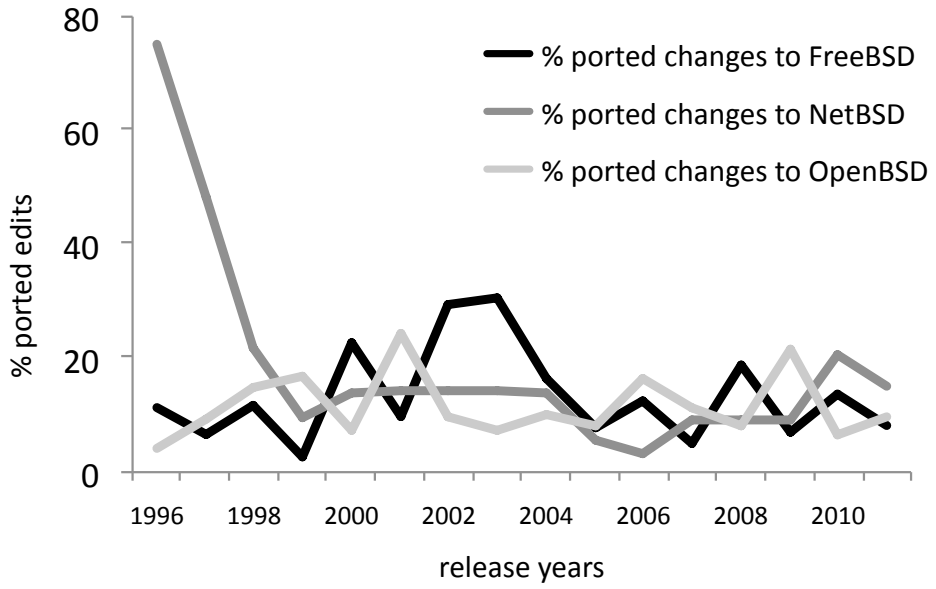
lease i. Based on the assumption that the code changes made in the peer projects must be available first to be transferred to another project, we compare these patches with $\Delta NetBSD_{(i-1,i)}$ using REPERTOIRE and identify the number of code lines ported from peer projects in each patch.

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 3.2, 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 4.1 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.

Figure 4.1: The porting rates in the BSD family



		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 and column represents a source project.

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.

Table 4.2: Linear regression of porting rates over time

	(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 4.1. 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.

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

To understand the reliability of ported changes, we investigate whether ported edits need more bug fixes than non-ported edits. We establish the relationship be-

tween bug fixes and ported changes by searching for keywords, *'bug,' 'fix,'* and *'patch'* in change commit messages using a heuristic similar to Mockus and Votta [54]. 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 [84]. Spearman correlation measures the association between two ranked variables. To measure the correlation of a sample data set of size n, first the raw data X_i and Y_i are converted to the ranked data x_i and y_i. Then the Spearman rank correlation is calculated on the ranked data set as the covariance of the two variables divided by the product of their standard deviations.

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

We choose Spearman rank correlation because to find when X values are increasing whether Y values are increasing or decreasing, i.e., we check monotonic relationship between X and Y. Here, we are not interested to see how absolute X and Y values are associated.

Table 4.3: 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

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 [55], our results indicate that *ported changes are likely to be relatively more safe and reliable than non-ported changes.*

These results imply that developers port well-tested bug fixes and feature

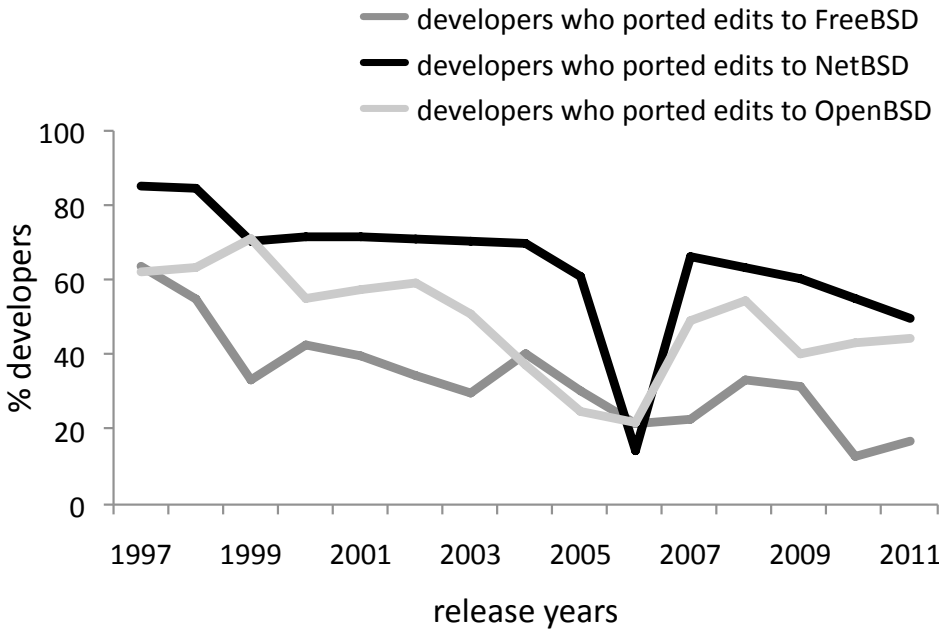
implementations from peer projects rather than risky and experimental features. This benefit of selective porting is aligned with Cordy's observation [19]. In financial software industry, copying an existing product to create a variant product is a recommended practice, because software forking allows developers to independently evolve product variants and reduces the risk of collective system failures caused by using a common platform.

4.2.3 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 ratio of the number of developers who ported edits in release i to the total number of active contributors of release i. Figure 4.2 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 [70]. A normalized

Figure 4.2: The percentage of developers who port changes from other projects per release



		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.

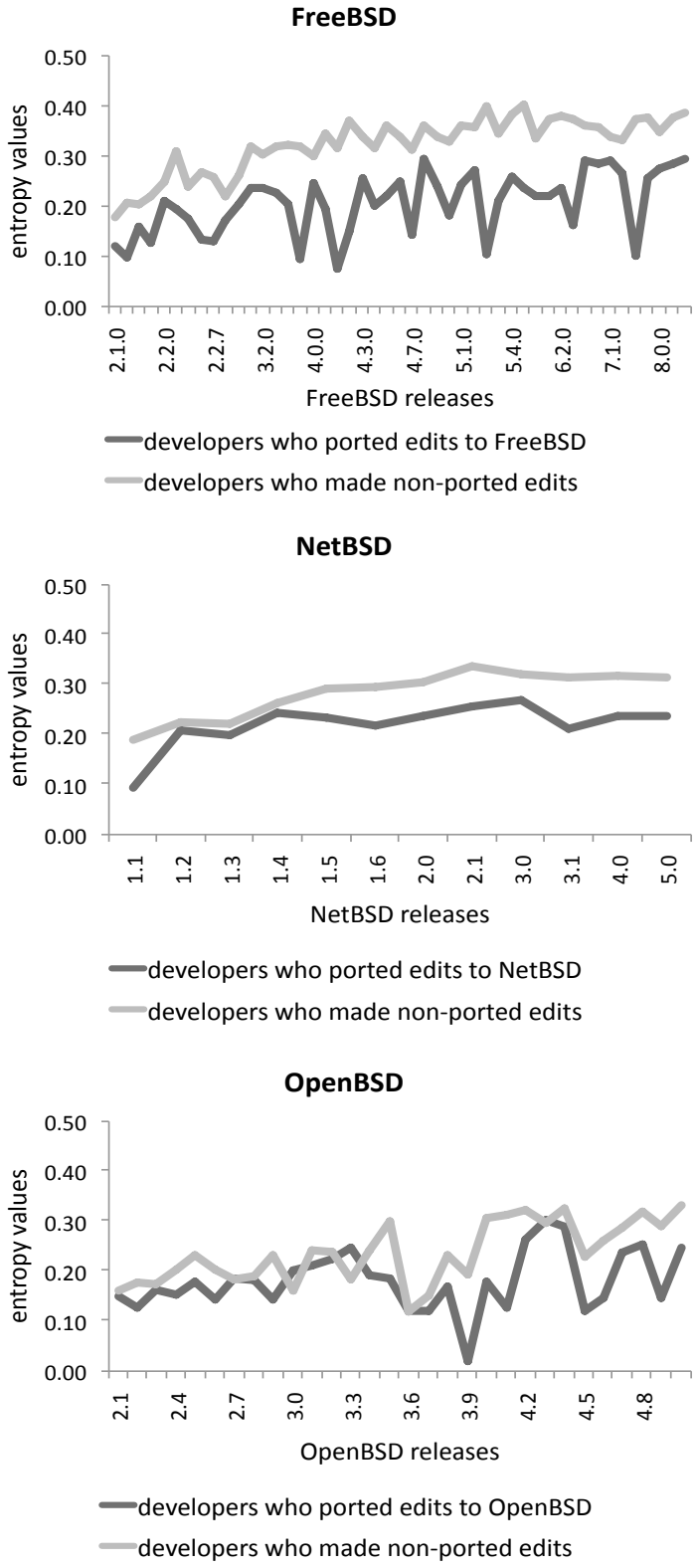


Figure 4.3: The workload distribution of developers of ported changes vs. non-ported changes in terms of entropy. X axis represents releases for each project.

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) [30] 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 4.3 shows 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.

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

To understand the efficiency of porting practice, we investigate how long it takes for a patch to propagate from one project to another. We measure porting latency as 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:

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

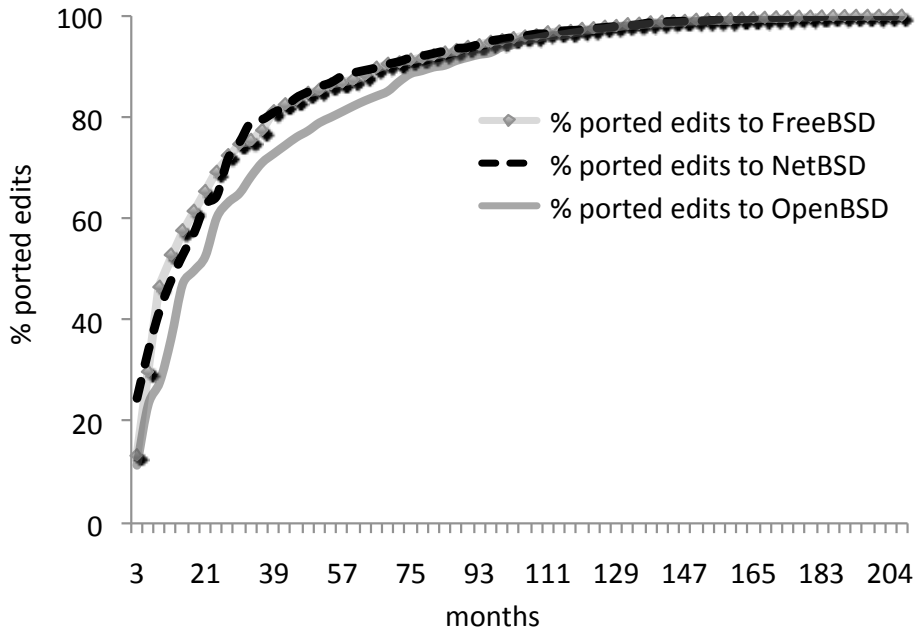
where $L(r)$ is the total number of ported lines of code in release r and N is the total number of releases in a project. It takes on average 734, 725, and 944 days to port an edit from other projects to FreeBSD, NetBSD, and OpenBSD respectively. Figure 4.4 shows a cumulative distribution of ported edits 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.

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.

4.2.5 Where is the porting effort focused?

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 of files with ported edits

Figure 4.4: The cumulative distribution of ported changes from other projects vs. patch propagation time.



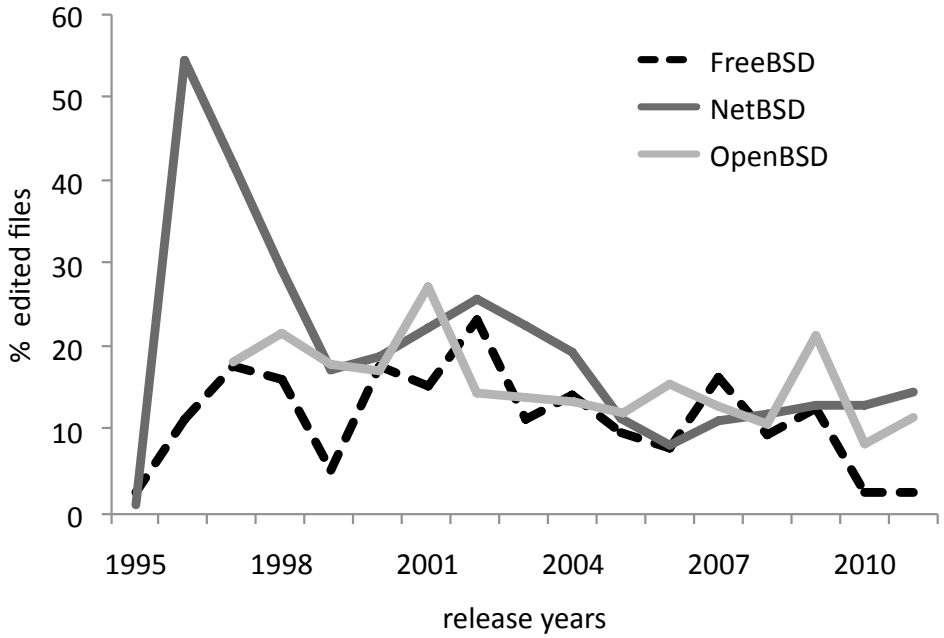
		Free Only	Net Only	Open Only	Both
FreeBSD	AVG		752	725	734
	MED		1787	1522	1668
NetBSD	AVG	807		676	725
	MED	1910		1640	1824
NetBSD	AVG	839	1056		944
	MED	1824	2000		1873

Days taken to port patches from other projects on average. Each row represents a target project. Each column represents a source project.

in release i divided by the total number of edited files in release i. Figure 4.5 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 4.5 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, 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 4.4 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% of total ported changes occurred 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.



		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 4.5: The percentage of edited files due to porting

Table 4.4: Top ten directories in individual BSD projects with the largest amount of 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%

The % values are the ratios of ported edits in the individual directories to all ported changes.

4.3 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`.

In terms of temporal granularity, we use program patches between each con-

secutive 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 4.2.2 indicate *only* correlation with defect density *not* causation.

External validity concerns the generalization of the findings. Our study focuses on FreeBSD, NetBSD and OpenBSD. We acknowledge that our case study on BSD may not generalize to other systems. For example, LibreOffice was forked from OpenOffice. While LibreOffice is an open source software, the later is maintained by Oracle Corporation. There might be limited code flow between the two office suites due to license disagreement. Nevertheless, we believe 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, and various distributions of Linux. We hope that other researchers replicate our results and thereby allow the community to build an empirical body of knowledge on the impact of forking and porting on various aspects like quality, dependencies, etc.

4.4 Summary

As a first step toward understanding the longitudinal impact of forking on maintainability and assessing whether forking is a sustainable practice, we applied our automated cross-system porting analysis tool, REPERTOIRE, to 18 years of parallel release history of the BSD product family and 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 the 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 finds 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 and an automated approach of notifying developers of potential collateral evolution.

Chapter 5

An Empirical Study of Porting Errors

Developers often port code from one context to another to introduce similar functionality or structural patterns [39]. They also port code across different projects in order to implement similar features or bug fixes, or to share similar API usage [10]. While porting, developers usually copy *reference code* and paste it to a *target* location and then adapt the pasted code to the surroundings [39, 50].

Existing studies show that developers often make mistakes while adapting ported edits to their surroundings and introduce errors in the codebase [35, 50]. Errors may also occur when two ported code fragments evolve differently. As a first step towards automatically detecting and diagnosing porting errors, we conduct an empirical study of porting errors documented in real world projects to better understand the extent and characteristics of porting errors found in practice. In this study, we focus on porting errors that arise when porting a patch to a similar, but not identical, context within the same project. We first identify porting errors that are reported and fixed by developers using the version histories from two large, open-source projects, Linux and FreeBSD. We then manually analyze these errors to understand the characteristics of the errors as well as the fixes. Most of the errors found in the artifacts used in our study can largely be characterized into five

categories including inconsistent control and data dependence, inconsistent variable and token renaming, and redundancy.

This chapter is organized as follows. First, in Section 5.1, we introduce several key terms. Next in Section 5.2, we present the study setup. Section 5.3 discusses the extent and characteristics of porting errors found in Linux and FreeBSD. Then in Section 5.4 and Section 5.5, we present a description of five categories of porting errors and their respective distribution. Finally, in Section 5.6 we discuss threats to the validity of this study and in Section 5.7 we conclude.

5.1 Definition

Definition 1. A *program patch*, $p := \Delta(v_1, v_2)$, is the set of syntactic program differences between two program versions, v_1 and v_2, where each element in the set is an atomic program statement that corresponds to an edit operation, e.g., insert, delete, move, and update.

Definition 2. *Ported code* is a pair of atomic program statements s_r and s_t in patches p_r and p_t respectively, such that s_r and s_t are syntactically similar, and also edited similarly.

Definition 3. *Context* of ported code is the set of program statements in a method that are not part of the ported code.

5.2 Study Method

To understand the extent and characteristic of porting errors in practice, we analyze the version histories for Linux and FreeBSD. Developers often document fixes to porting errors in commit messages. To detect how many bug fixes are related to porting, we find commit logs that contain at least one porting related keyword: `copy`, `cut`, `paste`, or `porting`, and at least one error related keyword: `error`, `bug`, `mistake`, `fix`, or `defect`. For example, a detected commit message in FreeBSD is "`Fix cut&paste bug which would result in a panic...`" . The corresponding code patch fixes the porting error.

In order to understand the nature of porting errors, we work backwards from a porting error fix by extracting three patches: (a) the fix patch, p_f, where the porting error is fixed, (b) the target patch, p_t, where the porting error is introduced into the codebase, and (c) the reference patch, p_r, which contains edits that serve as the template for the ported code. A fix patch p_f is the program patch associated with the mined commit message. For example, the fix patch corresponding to the commit message shown above, is represented by the colored lines in the Type-B1 example in Table 5.4. From the program locations edited in p_f, we use `cvs annotate` or `git blame`, to identify the target patch, p_t, which introduced the porting error. This process is similar to how Sliwerski et al. [74] identify a fix-inducing patch. We then use the REPERTOIRE tool to identify a set of candidate reference patches that may serve as the template for the target patch p_t [67]. The reference patch, by definition, has a commit date prior to the revision date of a target patch; hence, we consider patches available until the target patch date as

candidate reference patches. Finally, we select the reference patch, p_r, through a manual inspection of the possible candidates. For example, in the Type-B1 example in Table 5.4 where the developer forgot to update an identifier `bp` to `rabp` after porting code fragments from a reference patch, we expect the reference patch to contain the unaltered code fragment related to `bp`. When multiple patches contain similar unaltered code fragments, we select a patch with the maximum number of similar lines.

Using this method, we investigate porting errors in two large projects, Linux and FreeBSD. Table 5.1 shows the size of the two projects in KLOC, the duration of version histories under study, and the number of developers who committed changes during the period. In total, we find 113 porting errors in FreeBSD and 182 porting errors in Linux as shown in Table 5.2. Appendix B.1 describes the commit logs and the location of porting errors identified from FreeBSD.

Table 5.1: Study Subjects

	KLOC	**authors**	**years**
FreeBSD	4,479	405	18
Linux	14,998	6,839	3

5.3 Porting Error Fix Time and Developer Characteristics

To understand how often porting errors occur in practice and the overhead of fixing these errors, we investigate the following three research questions.

- **RQ1. How many porting errors are found from version histories?** Using the method described in the previous section, we find 113 and 182 porting

errors in FreeBSD and Linux over the last 18 and 3 years of their evolution history respectively.

- **RQ2. How long does it take to fix porting errors?** In order to understand how quickly porting errors are fixed, we measure the time gap between the introduction and the correction of the porting error. Table 5.2 summarizes the average time taken to fix a porting error. Though some of these bugs are fixed quite fast, say within the same day, the median values—39 days for FreeBSD and 181 days for Linux—show most porting errors take more than a month to be detected and fixed. The long tail distribution of the bug-fix time shown in Figure 5.1 suggests some bugs take even up to 2 years to be resolved. In a fast evolving software, a longer correction time may indicate the bug is difficult to be detected and fixed suggesting a non-trivial time and effort is required to fix the porting error.

Table 5.2: Time required to fix porting errors

	bug count	**min**	**max**	**median**	**avg**
FreeBSD	113	0	3894	39	341.52
Linux	182	0	2720	181	428.52

- **RQ3. Are the creator and the resolver of a porting error the same people?** If porting errors are resolved by different people who did not commit the porting errors in the first place, the resolver of a bug must learn the other developer's code to fix the error properly. Fixing other developers' code may require non-trivial program understanding effort. 58% and 54% porting errors

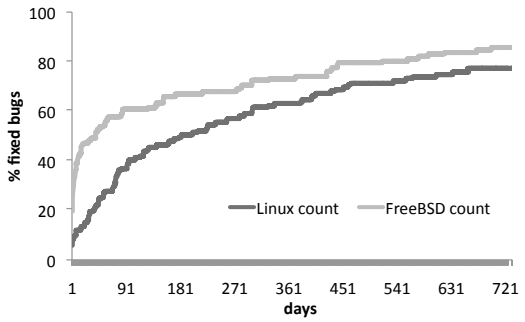


Figure 5.1: The cumulative distribution of the number of bug fixes and bug fix time.

in Linux and FreeBSD respectively were fixed by developers who did not introduce the errors.

These results indicate that the number of porting errors is significant in practice, and it takes non-trivial effort to detect and fix these porting errors. In fact, only in two months (between January 2013 and February 2013) there are already 8 fixes for porting related issues in Linux.

5.4 Types of Porting Errors in Practice

Based on the 295 porting errors from FreeBSD and Linux, we derive common types of porting errors. We examine the reference and the target contexts of the ported edits and the corresponding patches. Five common types of porting errors emerge from these examples. The following subsections describe each error type along with representative code snippets and log messages. For each example, we show the reference edits and target edits along with the subsequent fixes. The

ported lines start with "+". The errors are marked in red. The fixes are highlighted in **green**.

5.4.1 Type-A: Code is inserted into a different control flow context.

Many porting errors arise from edits that are ported to a different control flow context and are not adapted correctly with respect to the context. In the Type-A example shown in Table 5.3, there is an extra `for loop`, highlighted in gray, in the reference context. Thus, the `continue` statement in the reference is intended to match the inner `for loop`. In the target context, however, there is only one `for loop`. Thus, the `continue` statement (marked in red) unintentionally matches the wrong `for` loop. The corresponding fix patch removes the `continue` statement in the target context to fix the error.

Table 5.3: An Example of Type-A inconsistency

FreeBSD commit: `src/sys/kern/sched_4bsd.c`
version 1.90, Author: davidxu, Date: 2006/11/14
Log: Fix a *copy-paste bug* in NON-KSE case.

Reference File: `src/sys/kern/sched_4bsd.c`

```
FOREACH_KSEGRP_IN_PROC(p, kg) {
    awake = 0;
    FOREACH_THREAD_IN_GROUP(kg, td) {
        ...
+       if (ke->ke_cpticks == 0)
+           continue;
        ...
+        if(FSHIFT >= CCPU_SHIFT) {
+            ke->ke_pctcpu += (realstathz == 100)
+                ? ((fixpt_t) ke->ke_cpticks) <<
+                (FSHIFT - CCPU_SHIFT) :
+                100 * (((fixpt_t) ke->ke_cpticks)
+                << (FSHIFT - CCPU_SHIFT)) / realstathz;
        }
        ...
    }
...
}
```

Target File: `src/sys/kern/sched_4bsd.c`

```
FOREACH_THREAD_IN_PROC(p, td) {
   awake = 0;
   ...
+  if (ke->ke_cpticks ==!= 0)
+              continue;
   {
   ...
+   if(FSHIFT >= CCPU_SHIFT) {
+        ke->ke_pctcpu += (realstathz == 100)
+            ? ((fixpt_t) ke->ke_cpticks) <<
+            (FSHIFT - CCPU_SHIFT) :
+            100 * (((fixpt_t) ke->ke_cpticks)
+            << (FSHIFT - CCPU_SHIFT)) / realstathz;
    }
   ...
}
```

5.4.2 Type-B: Forget to adapt identifiers (variables, types, constants) to the target context.

Developers often forget to adapt variable, type, and constant names according to the target context and these inconsistent renamings lead to porting errors. This type of porting error is further split into two sub-categories:

Type-B1: Inconsistent renamings. Developers rename some occurrences of an identifier i, but forget to update all occurrences of the identifier i consistently. For example, pointer `bp` is updated to pointer `rabp` three times, missing the instances marked in red in the Type-B1 example in Table 5.4.

Table 5.4: An Example of Type-B1 inconsistency

FreeBSD commit: `src/sys/kern/vfs_bio.c`, version 1.351
Author: phk, Date: 2003-01-05
Log: Fix *cut&paste* bug which would result in a panic because buffer was being biodone'ed multiple times.

Reference File: `src/sys/kern/vfs_bio.c`

```
+ if ((bp->b_flags & B_CACHE) == 0) {
+  if (curthread != PCPU_GET(idlethread))
+    curthread->td_proc->p_stats->p_ru.ru_inblock++;
+  bp->b_iocmd = BIO_READ;
+  bp->b_flags &= ~B_INVAL;
  ...
+  if (vp->v_type == VCHR)
+    VOP_SPECSTRATEGY(vp, bp);
+  else
+    VOP_STRATEGY(vp, bp);
  ...
 }
```

Target File: `src/sys/kern/vfs_bio.c`

```
+ if ((rabp->b_flags & B_CACHE) == 0) {
+  if (curthread != PCPU_GET(idlethread))
+    curthread->td_proc->p_stats->p_ru.ru_inblock++;
+  rabp->b_flags |= B_ASYNC;
+  rabp->b_flags &= ~B_INVAL;
  ...
+  if (vp->v_type == VCHR)
+    VOP_SPECSTRATEGY(vp, bp rabp);
+  else
+    VOP_STRATEGY(vp, bp rabp);
...
}
```

Type-B2: Incomplete renamings of related identifiers. Developers consistently rename an identifier, but forget to update all related identifiers. In the Type-B2 example in Table 5.5, all instances of the `OFDM` related macro `IWL_FIRST_OFDM_RATE` are updated to the CCK related macro `IWL_FIRST_CCK_RATE`. How-

ever, the variable `ofdm` and the related macro `lowest_present_ofdm` are not updated to `cck` and the related macro `lowest_present_cck`. The corresponding fix patch replaces the token `ofdm` with the token `cck` to fix this error.

Table 5.5: An Example of Type-B2 inconsistency

Linux commit: 5edd0b946a0afeb1d0364a3654328b046fb818a2
Author: Emmanuel Grumbach, Date: 2013-11-20
Log: Fix a copy paste error in iwl_calc_basic_rates which leads to a wrong calculation of CCK basic rates.

Reference File: `../wireless/iwlwifi/dvm/rxon.c`

```
...
+if (IWL_RATE_24M_INDEX < lowest_present_ofdm)
+     ofdm  |= IWL_RATE_24M_MASK >> IWL_FIRST_OFDM_RATE;
+if (IWL_RATE_12M_INDEX < lowest_present_ofdm)
+     ofdm  |= IWL_RATE_12M_MASK >> IWL_FIRST_OFDM_RATE;
...
```

Target File: `../wireless/iwlwifi/dvm/rxon.c`

```
...
+ if (IWL_RATE_11M_INDEX < lowest_present_ofdmcck)
+     ofdmcck |= IWL_RATE_11M_MASK >> IWL_FIRST_CCK_RATE;
+ if (IWL_RATE_5M_INDEX < lowest_present_ofdm)
+     ofdmcck |= IWL_RATE_5M_MASK >> IWL_FIRST_CCK_RATE;
...
```

5.4.3 Type-C: Code is inserted to a different data initialization context.

In the Type-C example in Table 5.6, the first argument of the `strcmp` method `optarg` is initialized differently in the reference and target edits. `optarg` is an environment variable initialized by the `getopt()` call that parses the command line arguments and stores the next argument to `optarg`. Hence, the function call `getopt()` and the use of variable `optarg` should occur as a pair. In the reference context, `optarg` is used after `getopt()` and thus is initialized properly.

In the target context, however, there is no call to `getopt()`. Thus, `optarg` is not initialized properly.

Table 5.6: An Example of Type-C inconsistency

FreeBSD commit: `rc/sbin/gpt/gpt.c`
version 1.16, Author: marcel, Date: 2006-07-07
Log: Fix cut-n-paste bug: compare argument s against known aliases, not the global optarg.

Reference File: `src/sbin/gpt/gpt.c`

```
main(int argc, char *argv[]) {
...
 while ((ch = getopt(argc, argv,...)) != -1)
   switch (ch) {
   ...
+    case 'o':
+      if (strcmp(optarg, "space") == 0) {
+        opt = FS_OPTSPACE;
         ... } ... }
```

Target File: `src/sbin/gpt/gpt.c`

```
parse_uuid(const char *s, uuid_t *uuid) {
  ...
  switch (*s) {
+    case 'e':
+     if (strcmp(optarg s, "efi") == 0) {
+        uuid_t efi = GPT_ENT_TYPE_EFI;
      ... } ... } }
```

5.4.4 Type-D: Redundant operations.

Developers may inadvertently introduce redundant operations when they port code to the wrong place, e.g., where it already performs the same operation, or they may not update ported edits correctly to ensure there are no redundant computations in the target context. In the Type-D example in Table 5.7, a code fragment related to `memcpy` was ported to the same function body twice under the same scope in FreeBSD. The corresponding patch removes `memcpy` and the `buffer`

initialization statements to correct the redundant operations.

Table 5.7: An Example of Type-D inconsistency

Linux commit: f9c2fdbab1f1854f2bfcc75c326d0f4537ec2a7e
Author: John W. Linville, Date: 2011-04-29
Log: Looks like a copy-n-paste error, identical lines are a few lines below the ones removed, ...

Reference File: `src/sys/dev/mxge/if_mxge.c`

```
memset(&tsf_tlv, 0x00, sizeof(struct
    mwifiex_ie_types_tsf_timestamp));
...
+ memcpy(*buffer, &tsf_tlv, sizeof(tsf_tlv.header));
+ *buffer += sizeof(tsf_tlv.header);
```

Target File: `src/sys/dev/mxge/if_mxge.c`

```
+ memcpy(*buffer, &tsf_val, sizeof(tsf_val));
+ *buffer += sizeof(tsf_val);

  memcpy(&tsf_val,bss_desc->time_stamp,sizeof(tsf_val));
  ..
+ memcpy(*buffer, &tsf_val, sizeof(tsf_val));
+ *buffer += sizeof(tsf_val);
```

5.4.5 Type-E: Others.

Other porting errors we identified include incorrect formatting, e.g., indentation, that does not match with the rest of the target code structure, or unadapted comments that do not describe the target code correctly. For example, in FreeBSD file `src/sys/geom/stripe/g_stripe.h`, version 1.3, a comment related to "`Concat Name`" was not updated to "`Stripe Name`".

5.5 Distribution of Porting Errors in FreeBSD and Linux

By manually inspecting the sets of reference patch, p_r, target patch, p_t, porting error fix patch, p_f, associated commit messages, and bug descriptions, we categorize the porting errors into the five categories described above. Table 5.8 shows a distribution of the 113 cases of FreeBSD and 182 cases of Linux across the five categories of porting errors. The results show that a majority of porting errors are due to inconsistent renaming of identifiers (Type-B)—47.78% and 40.66% in FreeBSD and Linux respectively. The errors related to control (Type-A) and data (Type-C) flow inconsistency make up more than 25% of the total porting errors. The rest of the errors are either due to redundant operations (Type-D)—12.39% and 25.82%, or wrong indentation and comments (Type-E)—24.78% and 13.74% in FreeBSD and Linux respectively.

Table 5.8: Distribution of porting errors

	A	**B**	**C**	**D**	**E**	**Total**
FreeBSD	9	54	32	14	28	113
	7.96%	47.78%	28.31%	12.39%	24.78%	
Linux	23	74	26	47	25	182
	12.64%	40.66%	14.29%	25.82%	13.74%	

The error categories are not mutually exclusive. For example, an identifier update error (Type-B) may also cause an inconsistent data initialization error (Type-C)—17.7% and 1.6% of the porting errors in FreeBSD and Linux respectively are both types B and C. An inconsistent identifier update (Type-C) may also generate redundant operations (Type-D)—1.8% in FreeBSD and 2.7% in Linux. Sometimes, an inconsistent control flow (Type-A) may initialize the data erroneously (Type-

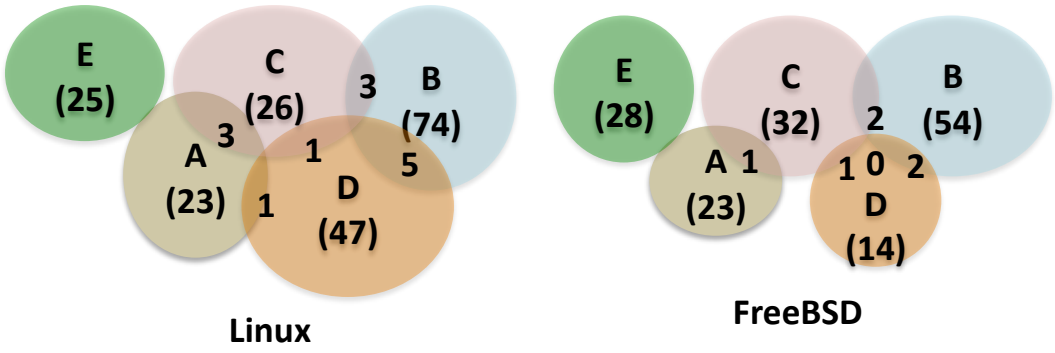


Figure 5.2: Relationship between different types of porting errors

C)—0.9% in FreeBSD and 1.6% in Linux. Figure 5.2 shows the distribution of the five porting error types in FreeBSD and Linux.

These results indicate that porting errors occur due to some common mistakes in adapting or updating ported code. These identified error types are later used as a basis for automatically detecting and diagnosing porting errors in Chapter 6.

5.6 Threats to Validity

Construct Validity. We rely on the method of mining for porting error related keywords in the commit messages. It is possible that developers may not document porting errors in commit messages when fixing porting errors.

Internal Validity. We assume that porting mistakes happen due to poor adaptation or inconsistent update of the ported code, which may not be always true. For example, porting code to an unintended context may also introduce errors. The five types of common porting errors are derived from the analyzed data and thus are subject to the experimenter's interpretation or categorization bias.

External validity. We study porting errors in FreeBSD and Linux. Both of these projects are written in C. Thus our categorization of porting errors may be biased towards C language features. Also, we study porting bugs for sequential code structure. Our observations may differ when a code fragment is ported from a sequential to a multi-threaded project. In this case, developers may have to adapt concurrency feature to support multi-threaded environment. Though our results may not generalize to other systems, we believe our study of two long-surviving, large scale operating systems provides meaningful insights to porting errors for sequential programs.

5.7 Summary

As a first step toward detecting and diagnosing porting errors, we identify porting errors from version histories and study the types and characteristics of porting errors. Our study of FreeBSD and Linux finds that the number of porting errors identified from commit messages is not insignificant and developers spend on average 341 days in FreeBSD and 428 days in Linux to fix these porting errors. Furthermore, correcting these porting errors may require a developer to understand the patch that another developer ported from a different context. In fact, 58% and 54% of Linux and FreeBSD porting errors are fixed by developers who did not introduce the errors.

Based on the data collected, we derive five common types of porting errors: (1) porting code to a different control flow context, (2) inconsistent renaming of identifiers, (3) porting code to a different data flow context, (4) redundant opera-

tions, and (5) inconsistent updates of comments and indentation. Our study also finds that inconsistent renamings are the most common source of porting mistakes (48% in FreeBSD and 41% in Linux). Porting code to different control and data flow contexts contributes to more than 25% of the errors in both FreeBSD and Linux. Leveraging this categorization of common porting errors we design an algorithm in Chapter 6 to automatically detect and diagnose porting errors.

Chapter 6

SPA: Semantic Porting Analysis and Error Diagnosis

Leveraging the characterization of porting errors discussed in Chapter 5, we design and implement SPA, an algorithm to detect and characterize porting inconsistencies. SPA detects semantic inconsistencies that arise due to the interactions between program statements in the ported code and the program statements surrounding the ported code. SPA takes two code patches as input: a reference patch and a target patch, each of which characterizes the syntactic differences between two program versions (Ref_{old} and Ref_{new}), and (Tar_{old} and Tar_{new}) respectively. SPA analyzes the reference and target patches to identify the ported code, and then uses static control- and data-dependence analyses to identify the impact of the ported code on the reference and target contexts. Next, SPA compares the impact of the ported code on the reference and target semantics to detect and characterize porting inconsistencies. Developers may also benefit to know how these inconsistencies change the porting behavior. As part of future work, we also discuss how SPA can be extended to generate test cases illustrating inconsistent porting behavior.

To evaluate the accuracy of SPA, we perform an empirical evaluation on four large open-source codebases: FreeBSD, Linux, Eclipse CDT, and Mozilla, and compare its results with two state-of-the-art tools, DejaVu [28] and Jiang et

al.'s clone related bug detection tool [35]. The results of our study show that SPA identifies semantic porting inconsistencies with 65% to 73% precision and 90% recall and identifies inconsistency types with 58% to 63% precision and 92% recall on average. SPA outperforms two related error detection tools by 14% to 17% more precision.

This chapter is organized as follows. Section 6.1 presents a motivating example that we use to explain SPA's approach in Section 6.2. Section 6.3 presents an empirical evaluation of SPA's capability to detect and characterize porting inconsistencies. Section 6.4 sketches an approach of extending SPA to help developers understand differential behaviors between the reference and target contexts in a concrete manner. Finally, Section 6.5 summarizes the contribution of this chapter.

6.1 Motivating Example

We use the code in Table 6.1 as a running example, which is an adopted version of code fragments from FreeBSD. Code is ported from a reference method, `freebsd4_getfsstat`, to a target method, `osf1_getfsstat`. Lines marked with "+" are the ported code. The reference and target contexts are syntactically different. In `osf1_getfsstat`, the ported lines `T9` and `T10` appear after two `if` statements at lines `T4` and `T6`. No such `if` statements are present in `freebsd4_getfsstat`. Also, the variable `buf` is initialized at line `T12`. Thus, `T13` is in a different data initialization context in the target than its corresponding line `R6` in the reference.

Table 6.1: Motivating Example (adopted and simplified porting example taken from FreeBSD)

Ref_{new}	Tar_{new}
R1.**int** freebsd4_getfsstat(**int** flags, **int** bufsize, ostatfs osb) {	T1.**int** osf1_getfsstat(**int** flags, **int** bufsize, osf1statfs osb) {
R2. statfs buf = **null**;	T2. statfs buf = **null**;
R3. **int** error = 0;	T3. **int** error = 0;
	T4. if (flags == GETFSSTAT)
	T5. **return** 0;
	T6. **if** (flags == WAIT)
	T7. flags = MNT_WAIT;
	T8.
R4.+ **int** count = bufsize / ostatfs.sizeof();	T9.+ **int** count = bufsize / ~~ostatfs~~**osf1statfs**.sizeof();
R5.+ **int** size = count * statfs.sizeof();	T10.+ **int** size = count * statfs.sizeof();
R6.+ error = copyout(osb, buf, size);	T11. if(size > 0)
	T12. buf = new statfs();
	T13.+ error = copyout(osb, buf, size);
R7.	T14. error = copyout(osb, buf, size);
R8. **return** error;	T15. **return** error;
R9.}	T16.}

Edited lines in a new version w.r.t. to the old version are presented in dark background. The ported lines begin with +. The red lines are inconsistent statements detected by SPA.

6.2 SPA Approach

To detect semantic inconsistencies, SPA takes a reference and target patch as input. The key steps are as follows: (1) detect code which is ported between the two patches, (2) determine a one-to-one correspondence between ported code in the target and reference, and (3) compute statements in the context that may impact the behavior of the ported code or may be impacted by the ported code. SPA conservatively estimates the program statements that are impacted by ported code using intraprocedural control- and data-dependence analyses. (4) Mark the impacted statements in the target and reference that differ in program-flow and identifier names as inconsistent.

After detecting a porting inconsistency, SPA categorizes it according to the five error types: Types A, B1, B2, C and D, described in Chapter 5.[1] The final output is a collection of inconsistent program statements in Ref$_{new}$ and Tar$_{new}$: IC_{ref} and IC_{tar} respectively, and the types of the inconsistencies.

6.2.1 Identifying Impact of the Ported Code

We present the three main steps to identify the impact of the ported code. The inputs to SPA are two patches specifying the syntactic differences between Ref$_{old}$ and Ref$_{new}$ and between Tar$_{old}$ and Tar$_{new}$: $p_{ref} := \Delta$(Ref$_{old}$, Ref$_{new}$) and $p_{tar} := \Delta$(Tar$_{old}$, Tar$_{new}$) respectively.

[1] Type E (unadapted indentation or comments) is not included in the scope of our diagnosis as this requires textual or lexical analysis and does not involve the semantics of code fragments.

Step 1. Identify Edits in Reference and Target

Similar to the edit script generation algorithm by Meng et al. [52, 53], SPA computes the syntactic edit operations (*insert*, *delete*, *move*, or *update*) required on the abstract syntax trees (ASTs) to transform Ref_{old} to Ref_{new} and Tar_{old} to Tar_{new}. This algorithm is inspired by Sydit and LASE and is implemented by modifying their AST differencing algorithm. Meng et al.'s AST differencing algorithm is an extended version of ChangeDistiller [24], where edit operations are represented as follows:

- **Insert**(n, p, k): AST node n is inserted as k^{th} child of AST node p.
- **Delete**(n): AST node n is deleted from its parent node.
- **Move**(n, p, k): AST node n is deleted from its parent and inserted as k^{th} child of AST node p.
- **Update**(n, v): The label of AST node n is updated to v.

In Table 6.1, three edit (*insert*) operations are identified in the reference corresponding to statements {`R4`, `R5`, `R6`}. Five edit operations (*insert*) are identified in the target corresponding to {`T9`, `T10`, `T11`, `T12`, `T13`} . SPA uses the edit operations to genearate the *edited nodes* E_{ref} and E_{tar}, corresponding to Ref_{new} and Tar_{new} respectively. An *edited node* e_p is an AST node corresponding to a statement in a program patch p. The lines corresponding to the edited nodes are highlighted using a gray background in Table 6.1.

Step 2. Identify Ported Nodes

SPA determines the correspondence between statements in the ported code in the reference and the target. It is possible that when a developer adapts ported code from one context to another, she may also insert or delete additional code; hence, there may be edited nodes that do not correspond to ported code. A *ported node pair* is a pair of AST nodes (r, t), where $r \in E_{ref}$ and $t \in E_{tar}$, and r and t have a unique correspondence with each other. This unique correspondence is determined by a function `clone` that takes two arbitrary AST nodes as input and outputs $true$ if the AST node types are identical and their labels are also similar above a certain threshold based on bi-gram similarity [71]. A bi-gram similarity detects the ratio of the total number of bi-grams common between two strings to the average number of bi-grams representing the strings. The output ranges from 0 to 1. A high value indicates that strings are either identical or very similar i.e., when developers rename identifiers after porting. We set the similarity threshold to a high value of 0.8 to ensure that the matched labels are very similar to each other, indicating truly ported nodes. Our definition of ported node pair is very restrictive to reduce false positives in the later steps; we only consider one-to-one correspondences between a reference and a target node and ignore node pairs with one-to-many correspondences.

$$PM = \{(r, t) | r \in E_{ref} \wedge t \in E_{tar} \wedge clone(r, t)\} \tag{6.1}$$

PM is a set of ported node pairs where each pair $(r, t) \in PM$ represents

a node ported from a reference patch to a target patch as defined in Equation 6.1. Each node in the pair (r, t) is referred as a *ported node*. For example, $R5$ and $T10$ have the same AST node type (`declaration`) and label (`size= count + statfs.size()`), hence $clone(R5, T10)$ is `true` and $(R5, T10)$ is a ***ported node pair***. However, no AST nodes in E_{ref} are syntactically similar to node $T11$ in E_{tar}. Therefore, $T11$ is not a member of any ported node pairs. The statements corresponding AST nodes in PM are identified with "+" in Table 6.1.

Step 3. Identify Impacted Nodes

Next, SPA identifies the AST nodes in Ref_{new} and Tar_{new} that are either impacted by or impact the semantics of the ported nodes. The impacted nodes include all of the ported nodes, and the subset of the context nodes that may affect the porting semantics. SPA identifies the impacted nodes using static intraprocedural data- and control-dependence analyses [79] with respect to the ported nodes.

Data Dependence. Statement S_2 is *data dependent* on S_1, if S_1 defines a variable v and S_2 uses v, such that there exists a path from S_1 to S_2 along which v is not killed (redefined).

Control Dependence. Statement S_2 is *control dependent* on S_1, if execution of S_2 depends on the decision made at S_1.

Program Dependence Graph. A program dependence graph, $PDG := \langle DN, DE \rangle$, is a set of vertices DN representing program statements, and a set of edges, $DE \subseteq DN \times DN$, representing the control and data dependencies between statements.

A control dependence graph (CDG) is a sub-graph of a PDG, where the edges represent control dependencies between vertices (program locations), whereas a data dependence graph (DDG) is a sub-graph of the PDG, where the edges represent data dependencies between vertices.

In SPA, we construct the PDG vertices with AST nodes, each of which represents an atomic program statement, and the edges correspond to the control and data dependences between statements. The impacted nodes in Ref_{new} and Tar_{new} are derived from their respective program dependence graphs, PDG_{ref} and PDG_{tar}. Given a set of vertices mapping to ported nodes $V_p \subseteq \text{Ref}_{new}$ and the PDG for Ref_{new}, we generate the impacted nodes I_{ref}. The impacted nodes map to vertices in the PDG reachable from V_p along the control and data dependence edges. Similarly, we can find I_{tar} from $V_p \subseteq \text{Tar}_{new}$. The vertices, `T6` and `T7`, in Table 6.1 are not control or data dependent on ported code, hence not in the impact set.

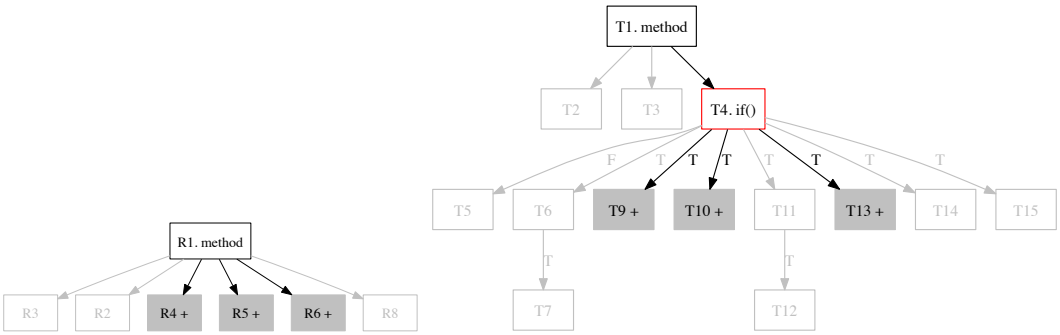


Figure 6.1: Control Dependence Graph w.r.t. the ported nodes in Table 6.1
Ported nodes are identified with '+'. Gray nodes are *isomorphic*. The faded vertices and edges have no control dependence on the ported nodes. The **Red** node is a control dependent inconsistency (Type-A) w.r.t. the isomorphic nodes.

6.2.2 Detecting and Categorizing Porting Inconsistencies

SPA detects and categorizes porting inconsistencies according to the types presented in Chapter 5, using ported node pairs, PM, impacted nodes, I_{ref} and I_{tar}, and the data- and control-dependence information computed in the previous steps.

Type-A: Code is inserted into a different control flow context

A Type-A inconsistency occurs when a ported node pair (r, t) has different control dependencies in their respective reference and target contexts. To detect this inconsistency, SPA performs the following steps:

- Given a pair of ported nodes, (r, t), we construct isomorphic sub-graphs starting from r in CDG_{ref} and from t in CDG_{tar}. A pair of vertices (v_r, v_t), where $v_r \in CDG_{ref}$ and $v_t \in CDG_{tar}$, is isomorphic if the (i) vertex labels have identical AST types and similar syntactic structures, and (ii) the vertices have the same relative position with respect to the ported nodes. We extend Komondoor et al.'s program slicing based clone detection algorithm [42] to construct the isomorphic sub-graphs.

- We detect inconsistent nodes in the context with respect to (r, t) and add them to the respective inconsistent sets, IC_{ref} and IC_{tar}. A node in I_{ref} (I_{tar}) is inconsistent if is reachable from r (t) in CDG_{ref} (CDG_{tar}), but, is not contained in the respective isomorphic subgraph.

Consider CDGs in Figure 6.1. Suppose that nodes $R4$ and $T9$ are a ported node pair. $R4$ is not control dependent on any node within the method body, while

$T9$ is control dependent on $T4$ along the $true$ control edge. $T4$ is then added to IC_{tar}, as it is reachable from ported node $T9$ yet does not have a corresponding node in the reference. Table C.1 in the appendix shows an example of Type-A inconsistency that SPA finds in the `Eclipes CDT` codebase and the individual inconsistency detection steps.

Type-B: Forget to adapt identifiers (variables, types, con- stants) to the target context.

A Type-B inconsistency occurs when developers forget to update identifiers to fit the target context correctly. To detect this inconsistency, we first construct the isomorphic sub-graphs on CDG_{ref} and CDG_{tar} with respect to the ported node pairs, as described earlier. For each isomorphic node pair in CDG$_{ref}$ and CDG$_{tar}$, we extract the corresponding identifiers, i.e., variables, types, and method names, and align them based on their syntactic similarity. For example, given two isomorphic nodes with labels '$a = b + c$' and '$x = y + z$', variable a is aligned with x, variable b is aligned with y, and variable c is aligned with z. We rank each identifier mapping with a confidence value based on the number of times the mapping is encountered. Using these alignments, we generate two identifier maps: (a) $IdMap_{ref}$, a map from each reference identifier to its corresponding target identifiers, and (b) $IdMap_{tar}$, a map from each target identifier to its corresponding reference identifiers. If a one to many or a many to one relation is found in the maps, then a Type-B inconsistency is detected. We consider identifier mappings with the lowest (or, all when there is a tie) confidence values as the incorrect mappings, and characterize the vertices in the isomorphic sub-graphs corresponding to the incorrect mappings

as inconsistent.

The following table shows an example of $IdMap_{ref}$ generated from Table 6.1. SPA generates a map entry (`osf1statfs` →`ostatfs`) from the method signatures and (`osf1statfs` →`osf1statfs`) from the isomorphic nodes $R4$ and $T9$. Since the reference variable `osf1statfs` maps to two target variables, `osf1statfs` and `ostatfs`, a Type-B inconsistency is detected. Table C.3 in the appendix shows an example of Type-B inconsistency that SPA finds in an adopted `FreeBSD` porting example.

Isomorphic Nodes	**Identifier Map** (IdMap$_{ref}$)
(R1,T1)	flags → flags (1), bufsize → bufsize (1) osf1statfs → ostatfs (1), osb → osb (1)
(R4,T9)	count → count (1) , bufsize → bufsize (2) osf1statfs → ostatfs (1), osf1statfs (1)

The inconsistent mapping is highlighted in red.

Sometimes developers forget to update *related identifiers*, as shown in the Type-B2 example in Chapter 5. To detect this inconsistency, we carry out a similar process at the granularity of tokens as opposed to identifiers after separating identifier names using separators '-', '_', or a camel case convention. For example, `OFDM` is mapped to `CCK` once, while `ofdm` is mapped to `ofdm` twice.

Type-C: Code is inserted to a different data initialization context.

A Type-C inconsistency occurs when a ported node pair (r, t) has different data dependencies in their respective reference and target contexts. This analysis is similar to our Type-A diagnosis but uses data dependence graphs (DDG) instead of CDGs. To detect this inconsistency, SPA works as follows:

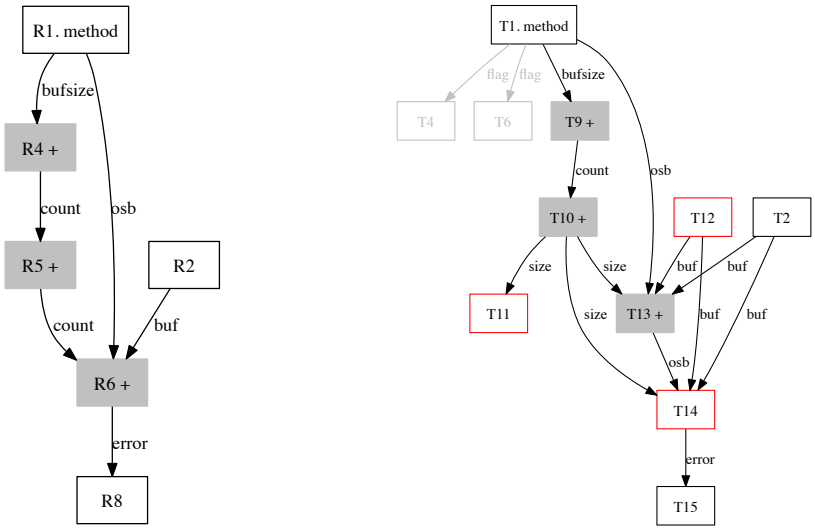


Figure 6.2: Data Dependence Graph w.r.t. the ported nodes in Table 6.1

Ported nodes are identified with '+'. Gray nodes are *isomorphic*. The faded vertices and edges have no data dependence on the ported nodes. The **Red** nodes are data dependent inconsistency (Type-C) w.r.t. the isomorphic nodes.

- Given a pair of ported nodes, (r, t), we construct isomorphic sub-graphs starting from r in DDG_{ref} and from t in DDG_{tar}. We follow a similar process to construct the isomorphic subgraphs from the DDGs as described in the diagnosis of Type-A inconsistencies.

- Detect inconsistent nodes in the context with respect to (r, t) and add to the respective inconsistent sets, IC_{ref} and IC_{tar}. A node in I_{ref} (I_{tar}) is inconsistent if it is reachable from r (t) in DDG_{ref} (DDG_{tar}), but is not contained in its respective isomorphic subgraph.

Figure 6.2 shows the DDGs corresponding to Table 6.1. Here, `R6` and `T13` for a ported node pair. In the reference implementation, `R6` is data dependent on `R2`

for the definition of variable `buf`. However, statement `T13` in the target implementation is data dependent on the definition of `buf` at `T2` and `T12`. Although `R2` and `T2` are isomorphic, the dependence on `T12` creates an additional data dependence in the target implementation that is not present in the reference implementation. Therefore, `T12` is added to IC_{tar}. Similarly, `R5` and `T10` form a ported node pair, and both have defined variable `size`. However, in the reference `size` is used by statement `R6`, while in the target `size` is used by statements `T11`, `T13`, and `T14`. Although `R6` and `T13` are isomorphic, `T11` and `T14` create an additional data dependence in the target implementation that is not present in the reference implementation. Therefore, the `T11` and `T14` are added to IC_{tar}.

Table C.5 in the appendix shows an example Type-C inconsistency that SPA finds in one of the `FreeBSD` porting example.

Type-D: Redundant operations.

To detect redundant ported code, where ported code is duplicated, SPA checks for pairs of vertices in CDG_{tar} that have identical labels and types and that are control dependent on the same impacted vertex. Note that we only look for a type-D inconsistency in Tar_{new}. In Table 6.2, statements `T13` and `T14` in the target implementation have identical syntax, and both are control dependent on the impacted statement `T4`. Thus, SPA characterizes the corresponding statements `T13` and `T14` as redundant. Table C.7 in the appendix shows an example Type-D inconsistency that SPA finds in one of the `FreeBSD` porting example.

Table 6.2 shows the nodes that are inconsistent with respect to the ported

Table 6.2: Characterization of Porting Inconsistencies in Table 6.1

Inconsistent Control Dependent Nodes (Type-A)	T4
Inconsistent Identifier Renaming (Type-B)	T9 (identifier: ostatfs)
Inconsistent Data Dependent Nodes (Type-C)	T11,T12,T14
Redundant Nodes (Type-D)	T13,T14

code in Table 6.1, along with their corresponding inconsistency types.

6.3 Experimental Results

In this section, we present an empirical evaluation of SPA's ability to detect and diagnose porting inconsistencies in FreeBSD, Linux, Eclipse CDT, and Mozilla. We compare the accuracy of the results computed by SPA with the results computed by two state-of-the-art tools, Jiang et al.'s clone related error detection tool [35] and DejaVu [28]. Jiang et al. model the context of ported code, in terms of their immediate preceding lines, even if the context does not have any control or data dependence on ported code. Though DejaVu extends Jiang et al. by refining clone detection results to determine ported code, it still suffers from the same limitation as Jiang et al., as the context is identified based on physical location proximity not based on control and data flow dependence on ported code.

We also compute SPA's accuracy to characterize potential inconsistencies based on the categories defined in Chapter 5. To this end we investigate two research questions:

- **RQ1.** Can SPA accurately *detect* porting inconsistencies?
- **RQ2.** Can SPA accurately *categorize* different types of porting inconsistencies?

6.3.1 Study Subjects

To evaluate SPA, we use porting examples from four different real-world artifacts: FreeBSD, Linux, Eclipse CDT, and Mozilla. Except for Mozilla, the reference and target patches for each artifact are computed using REPERTOIRE [66]. From these, we randomly select (a) 20 porting examples from FreeBSD, (b) 10 porting examples from Linux, (c) 60 porting examples from Eclipse CDT that are ported from CDT versions `CDT_2_0` to `CDT_8_1_1`, and (d) 42 Mozilla examples from the annotated data set of copy-paste errors provided by Gabel et al. [28]. The FreeBSD and Linux artifacts are from the data sets used in Section 5. To retrieve a large number of porting instances, we choose `CDT_2_0` and `CDT_8_1_1` versions which are 98 months apart. The Mozilla examples were obtained from DejaVu's annotated data set[2], because Dejavu is not an open-source tool. In the Mozilla examples, we treat an entire program as a program patch whose old version is empty, because SPA works on program patches as opposed to entire programs. We use a combination of commit logs and manual inspection to annotate the types of potential porting errors in selected target patches of the subject artifacts.

The current version of SPA analyzes only Java source code, so we convert the

[2] `http://wwwcsif.cs.ucdavis.edu/~gabel/research/dejavu_mozilla.zip`

C and C++ porting examples from Linux, FreeBSD, and Mozilla examples using a free C/C++ to Java code converter [1].

6.3.2 Methodology

SPA is implemented by extending LASE [53] and Sydit [52], which extract edit scripts to automate systematic program changes. SPA uses LASEs edit script generation algorithm to determine the syntactic ported edits. SPA also extends the control and data dependence analysis of Sydit to identify the impact of ported nodes in the reference and target programs respectively (see Section 6.2.1). This step bears resemblance to how Sydit identifies the context of edit operations using control and data analysis. Sydit's dependency analysis uses *crystal* [2], a static analysis framework to analyze Java source code. The Sydit implementation of the dependence analysis is in part reused in SPA.

We measure SPA's capability to detect and categorize porting errors in terms of precision and recall. For each error type e defined in Chapter 5, suppose that S is the set of examples where a porting inconsistency is detected by SPA and its error type is reported by SPA to be e. Suppose that A is the set of examples where a porting inconsistency is manually determined to be of type e. Then the precision and recall of SPA in categorizing porting inconsistencies are defined as follows:

Precision. the percentage of porting inconsistencies of type e found by SPA that are also known to be type e i.e., $\frac{|A \cap S|}{|S|}$

Recall. the percentage of the known inconsistencies of type e, which are

also found to be type e by SPA, i.e., $\frac{|A \cap S|}{|A|}$

To evaluate the accuracy of SPA's error detection capability, we calculate precision and recall without considering individual error types.

6.3.3 Study Results

Accuracy of Detecting Porting Inconsistencies. We compare SPA's ability to detect porting inconsistencies with Jiang et al.'s clone related bug detection algorithm [35][3] and DejaVu [28]. Table 6.3 summarizes the comparison of SPA with Jiang et al. using the Eclipse CDT artifact and with DejaVu on the Mozilla examples. The first row represents the number of potential porting errors, regardless of error type, that were detected by the respective tools. We also report the number of false positives, false negatives, precision, and recall of the error detection capability of each tool. The results of our study show that SPA improves the error detection capabilities over Jiang et al. SPA improves the precision from 48% to 65%, and improves the recall from 87% to 90%.

Out of the 42 randomly selected examples from the DejaVu annotated Mozilla data set, our manual inspection shows that only 25 of them contain true porting inconsistencies. Thus, DejaVu's precision is 59.52%. For the same data set, SPA reports inconsistencies for 34 examples. Thus, SPA's precision in detecting errors on the Mozilla data set is 73.53% as shown in Table 6.3. Because this data set does not contain any examples where DejaVu fails to report an inconsistency, we are un-

[3]Jiang et al.'s clone detector Deckard and the associated clone bug detector were downloaded from `https://github.com/skyhover/Deckard`.

able to assess the number of false negatives for both DejaVu and SPA. Furthermore, because our comparison is limited to the data set where DejaVu already found porting inconsistencies, the precision of SPA could be lower if the comparison was done on a different data set.

We find that SPA reduces false positives over Jiang et al.'s tool and DejaVu in 14 and 8 cases respectively. For example, consider a case when a variable is initialized differently in the reference and target contexts. Later, both the reference and the target contexts reinitialize the variable in the same manner before using it in the ported code. In this case, SPA correctly does not report any inconsistency unlike other tools, because there is no data flow between the inconsistent initialization and the ported code.

The cases where all the three tools incorrectly detect inconsistencies include porting code from a `while` context to a `for` context and from a `if` context to a switch-case context etc.

Table 6.3: Inconsistency detection results for Eclipse CDT and Mozilla

	Eclipse CDT		Mozilla	
	SPA	**Jiang's Tool**	**SPA**	**DejaVu**
Detected	43	56	34	42
False Positive	15	29	9	17
False Negative	3	4	-	-
Precision	65.11%	48.21%	73.53%*	59.52%*
Recall	90.32%	87.09%	-	-

*The comparison is done on the data set where DejaVu already reported porting errors.

Accuracy of Categorizing different types of Porting Inconsistencies. Table 6.4 shows the precision and recall for SPA in categorizing potential porting errors in

Table 6.4: Inconsistency characterization results on FreeBSD and Linux

	A	**B1**	**B2**	**C**	**D**
SPA Detected	10	8	6	9	5
From commit logs	5	8	5	6	8
Precision	50%	87.5%	66.66%	66.66%	100%
Recall	100%	87.5%	80%	100%	62.5%
Manually annotated	7	8	5	8	8
Precision	70%	87.5%	66.66%	87.5%	100%
Recall	100%	87.5%	80%	100%	62.5%

FreeBSD and Linux for the error types A, B1, B2, C, and D. SPA has precision ranging from 50% for Type A to 100% for Type D. The recall for SPA ranges from 62.5% for Type D to 100% for Type A and Type C w.r.t. the porting errors reported in the version histories (see 2^{nd} row in Table 6.4). Version history based evaluation is often conservative in the sense that when there is no mention of porting errors in the commit messages, it does not necessarily imply the absence of porting inconsistencies. To overcome this limitation, we compare SPA results against the type and location of inconsistencies that were identified through manual inspection of individual patches. The comparison against this annotated set is shown in Rows 5-7 in Table 6.4.

Table 6.5 and 6.6 summarize the number of porting inconsistencies for each error type, and the precision and recall based on the manually identified error types for Eclipse CDT and Mozilla data sets. In Eclipse CDT, SPA detects and characterizes 62 porting inconsistencies—77% are Type-A, 16% are Type-B1, 12% are

Table 6.5: Inconsistency diagnosis results for Eclipse CDT

	A	**B1**	**B2**	**C**	Total
SPA Detected	33 (53%)	7 (11%)	5 (8%)	17 (27%)	62
Annotated	23	7	4	5	39
False Positive	12	2	2	12	26
False Negative	2	2	1	0	3
Precision	63.63%	71.43%	60%	29.41%	58.06%
Recall	91.30%	71.43%	75%	100%	92.31%

we do not detect any type-D inconsistency here.

Table 6.6: Inconsistency diagnosis results for Mozilla

	A	**B1**	**B2**	**C**	Total
SPA Detected	15(28%)	12 (22%)	4 (7%)	23 (43%)	54
Annotated	13	6	2	13	34
False Positive	2	6	2	10	20
False Negative	0	0	0	0	0
Precision	86.66%	50.0%	50.0%	56.52%	62.96%
Recall	100%	100%	100%	100%	100%

we do not detect any type-D inconsistency here.

Type-B2, and 40% are Type-C. In Mozilla, SPA detects 54 instances of porting inconsistencies, of which 28%, 22%, 7%, and 43% are of Type A, Type B1, Type B2, and Type C respectively. No Type-D inconsistency is reported in these two data sets. On average, SPA achieves 58% precision and 92% recall in Eclipse CDT, and 63% precision and 100% recall in Mozilla data set.

In detecting Type-A inconsistencies, SPA may report false positives when, for example, code is ported from a `for` block to an equivalent `while` block, because these two loops have different syntaxes. SPA may generate a false positive of Type-B1 when the relative ordering of program variables is changed, but the semantic remains unchanged, e.g., a statement x = x+y in the reference implementation is modified to x = y+x in the target. When characterizing Type-B2 inconsistencies, SPA may report false positives when, for instance, the names cannot be tokenized properly due to inconsistent naming conventions. For example, if a ported node pair contains the variables `fooBar` and `foobar`, SPA correctly splits the first one into `foo` and `Bar` but does not split `foobar`. Thus, SPA misaligns the tokens. In case of Type-C inconsistencies, SPA may report a false positive when, for example, a variable is declared and defined in a single program statement in the reference, but the declaration and definition are separate statements in the target. Here, SPA reports an inconsistency because the AST node types are different (declaration versus assignment). With respect to false negatives, SPA is not able to detect redundancies that require a deeper semantic analysis, such as redundant `locking` calls in a concurrency construct.

In spite of these limitations, here are some success stories. A bug was fixed

in FreeBSD source file: `src/sys/dev/mxge/if_mxge.c`, version 1.27, with a commit message: *"Fix an mbuf leak caused by a cut&paste bug where the small ring's mbufs were never freed, but the big ring was freed twice"*. A buffer `rx_big` was mistakenly freed twice. SPA detects this bug successfully and categorizes it as a Type-D redundancy bug, which is also confirmed by the developers. Jiang et al's tool is not able to detect this bug since it does not handle redundancy. It took 26 releases and 432 days to detect and fix the error.

Another identifier renaming bug was fixed in Linux at commit id 2b9460. Code was ported from method `mlx4_ib_post_send` to `mlx4_ib_post_recv`, but variable `send_cq` was never updated to `recv_cq`. This bug caused a queue overflow in the `infiniband` driver module (a high-speed network driver) and took 974 days to fix. SPA successfully detected this error. This error was not detected by other tools because they do not check whether related variables were updated consistently (Type-B2).

6.4 Discussion: An Idea of Extending SPA to Assist Developers in Understanding Differential Behavior

Once SPA identifies program statements that may cause inconsistencies in porting behavior, developers may want to know how these inconsistencies change the semantics of the ported code in the target implementation w.r.t. the reference implementation. Here, we sketch an algorithm to generate test cases illustrating the inconsistent porting semantics. This algorithm could help developers understand the differences more concretely.

Using the information of ported nodes and inconsistent nodes identified by SPA, we could generate test cases in three steps. In the first step, we will identify program paths in the reference and target methods that may illustrate differential porting semantics using control- and data-dependence analysis. Second, using a symbolic execution technique, we will characterize the differential behavior in terms of *path-effect conditions*—constraints on program inputs and effects on program states [62]. Third, using a multi-staged equivalence checker, we will compare these differential behaviors of the ported code in its reference and target implementations. The final output should be a set of test cases that could illustrate the inconsistent behavior of the ported code.

6.4.1 Motivating Example

In this section, using a running example, shown in Table 6.7, we sketch how the test generation process works. The ported lines start with +. The control- or data- dependent statements w.r.t. the ported lines are underlined. Among them, red lines (`R7`, `R19`, `R21`, and `T10`) indicate inconsistent statements that are identified by SPA. However, not all of them illustrate differential behaviors. For example, in `ref.java` the ported lines `R22` to `R24` can only be reached if `R21` is `true`, i.e., buf $\neq$ null. The variable `buf` is not null if `R16` is `true`, i.e., `size` $>$ `0`. Similarly, ported code `T12` to `T14` can only be reached if `T10` is true, i.e., `size` $>$ `0`. Thus, though `R20`, `R21`, and `T10` share different control and data dependencies w.r.t. the ported lines, they essentially have identical effects on the ported line pairs (`T18`, `R10`) and (`T19`, `R11`) respectively. The actual inconsistency arises from an

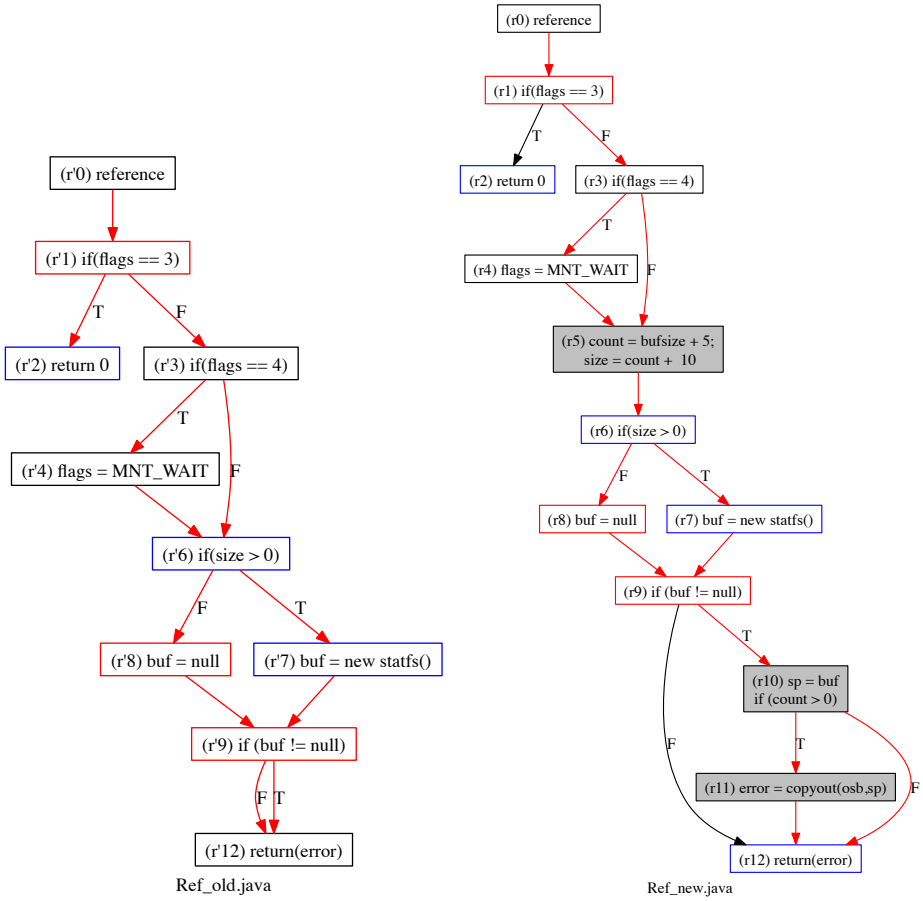
Table 6.7: Motivating Example illustrating test generation procedure

Reference File: ref.java	Target File: tar.java
R1.int reference(int flags, int bufsize, osf1statfs osb) {	T1.int target(int flags, int bufsize, ostatfs osb) {
R2. statfs buf;	T2. statfs buf;
R3. statfs sp;	T3. statfs sp;
R4. int count, size = 0;	T4. int count, size = 0;
R5. int error = 0;	T5. int error = 0;
R6.	T6.
R7. if (flags == 3)	T7.+ count = bufsize + 5;
R8. return 0;	T8.+ size = count + 10;
R9.	T9.
R10. if (flags == 4)	T10. if (size > 0) {
R11. flags = MNT_WAIT;	T11. buf = new statfs();
R12.	T12.+ sp = buf;
R13.+ count = bufsize + 5;	T13.+ if (count > 0)
R14.+ size = count + 10;	T14.+ error = copyout(osb, sp);
R15.	T15. }
R16. if(size > 0)	T16.
R17. buf = new statfs();	T17. return (error);
R18. else	T18.}
R19. buf = null;	
R20.	
R21. if (buf != null) {	
R22.+ sp = buf;	
R23.+ if (count > 0)	
R24.+ error = copyout(osb, sp);	
R25. }	
R26.	
R27. return (error);	
R28.}	

The ported lines begin with +. The underlined lines are statements impacted by the ported lines. The red lines are inconsistent statements found by SPA.

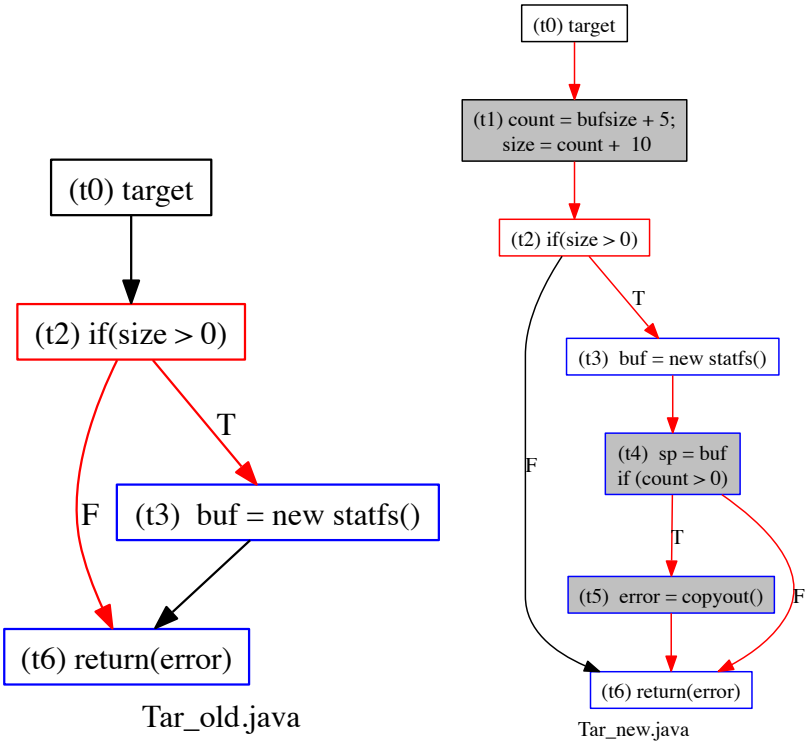
extra control predicate at R7. We now explain how we plan to generate test cases in order to help programmers understand differential behavior due to R7. In this thesis we sketch this algorithm, which could be implemented in future to support developers in examining the inconsistencies detected by SPA.

Figure 6.3: Impact Analysis and Path Selection on Ref_{new} and Ref_{old}



Ported nodes are marked in gray. Impacted nodes are marked in red or blue. Among the impacted nodes, red are identified as inconsistent by SPA analysis. The paths highlighted in red contain both ported and inconsistent nodes.

Figure 6.4: Impact Analysis and Path Selection on Tar_{new} and Tar_{old}



Ported nodes are marked in gray. Impacted nodes are marked in red or blue. Among the impacted nodes, red are identified as inconsistent by SPA analysis. The paths highlighted in red contain both ported and inconsistent nodes.

6.4.2 Test Generation Steps

The inputs to the test generation process are four program versions: Ref_{old}, Ref_{new}, Tar_{old}, and Tar_{new}. It also takes the output of SPA: a set of ported nodes (PM), a set of identifier mapping (IdMap_{ref}) that maps reference identifiers to target identifiers, and sets of *inconsistent nodes* with respect to the ported nodes, IC_{ref} and IC_{tar} respectively. The outputs are test cases that could concretely illustrate differential behavior with respect to the ported nodes.

Step 1. Identification of Inconsistent Program Paths

The goal of this step is to identify the program paths that may change the porting semantics in the reference and target implementations. Since porting semantics may differ due to the inconsistent nodes identified by SPA in Section 6.2, in this step we select program paths that contain the inconsistent nodes (IC_{ref} and IC_{tar}).

(1) Analysis on the new program versions Tar_{new} and Ref_{new}. Using static control- and data dependence analysis on the program's byte code, we identify nodes impacted by the ported nodes. These *impacted nodes* either control the program execution of the ported nodes, or change the values of variables read or written by the ported nodes. Thus, these impacted nodes indirectly determine the paths affected by the ported nodes in the new program versions. However, not all these impacted paths are reachable to and from inconsistent nodes. Since we are only interested in analyzing inconsistent porting behavior, we choose the impacted paths that contain at least one ported node and at least one inconsistent node. This helps

us to focus on the differential behavior involving ported nodes.

In `Ref_new.java` of Figure 6.3, the gray nodes $\{r5, r10, r11\}$ are ported nodes. $\{r1, r6, r9\}$ and $\{r5, r7, r8, r12\}$ are control and data dependent on the ported nodes respectively. These nodes are marked in blue or red. The red nodes correspond to the inconsistent nodes identified by SPA. From the paths covering impacted nodes, we select paths containing at least one ported node and at least one inconsistent node, because these are the only paths that could potentially change the porting semantics. For example, the impacted path `r9` to `r12` does not contain any ported node. Hence, this path is pruned. The red paths indicate the inconsistent paths selected in this step. Similarly, `Tar_new.java` of Figure 6.4 show the inconsistent paths in red.

(2) Analysis on the old program versions Tar_{old} and Ref_{old}. Some of the inconsistent paths identified from the new program versions may have existed in the old program versions. Developers may not be interested in such pre-existing differences as these differences do not arise due to porting. Thus, we treat such pre-existing inconsistencies separately. In order to identify this differing behavior, we detect a subset of inconsistent nodes that are also present in the old versions. We call them IC_{RO} and IC_{TO}. By definition, the inconsistent nodes in the old versions cannot be ported nodes in the new version, i.e., $\forall$ n $\in$ IC_{RO} : n $\notin$ PM_{ref}. Similarly, $\forall$ n $\in$ I_{TO} : n $\notin$ PM_{tar}.

By running control and data flow analysis of Tar_{old} and Ref_{old}, we select the program paths that are reachable to and from IC_{TO} and IC_{RO} respectively. These paths characterize the differential program behavior existed in the reference and

target programs even before ported edits. Figure 6.3 shows the selected paths in `Ref_old.java` (marked in red) covering the inconsistent nodes $\{r'1, r'8, r'9\}$.

Step 2. Generating Path Effect Conditions

For each path selected in Step 1, we compute the input constraints to execute this path and the program states after its execution. This is represented by partition-effect constraints [62], as defined in Table 6.8. In Table 6.9, π_{RO} and π_{TO} represent the partition-effect constraints caused by pre-existing differences in the old versions. π_{RN} and π_{TN} represent the partition-effect constraints caused by ported nodes and inconsistent nodes in the new versions.

In practice, Step 1 and Step 2 can occur together. DiSE can be extended to select inconsistent paths and generate path-effect constraints corresponding to those paths using an incremental symbolic execution approach [63]. While exploring a program path starting from a conditional predicate, we will check whether the path is inconsistent according to Step 1. If not, we will prune that path and backtrack to the conditional predicate. Also, we will not generate path constraints corresponding to the pruned path. For example, in `Ref_new.java` of Figure 6.3, while exploring the path from the conditional node `r9` to node `r12`, no inconsistent or ported nodes are encountered. Hence, we prune the path from `r9` to `r12` and do not generate the corresponding path constraint (`buf == null`). The final outcome will be a set of partition-effect constraints that covers only the inconsistent paths.

Table 6.8: Program behavior analysis in terms of partition-effect behavior: definitions adopted from Person et al. [62]

Partition-Effects Pair. A partition-effects pair, (i, e), consists of: an input constraint, i, which is a conjunction of relational expressions defined over constants and symbolic variables, and an effects constraint, e, which is a conjunction of expressions that equate written locations to expressions defined over constants and symbolic variables.

Symbolic Summary. A symbolic summary, for a method m, is a set partition-effect pairs $m_{sum} = \{(i_1, e_1), (i_2, e_2), ..., (i_k, e_k)\}$, where the input constraints are disjoint, i.e., $\forall\, 1 \leq j \leq k \forall 1 \leq j' \leq k' \wedge j \neq j' : i_j \wedge i'_j$ is unsatisfiable.

Table 6.9: Partition-Effect constraints for the target and reference programs corresponding to examples of Table 6.7.

Identifier mapping from the reference to target context (IdMap$_{ref}$)
bufsize $\rightarrow$ bufsize ; flags $\rightarrow$ flags
Path constraints in Ref$_{new}$ covering ported edits and inconsistencies (π_{RN})
R1: (bufsize + 5 $>$ 0) $\wedge$ (bufsize + 5 + 10 $>$ 0) $\wedge$ (flags == 4) $\wedge$ (flags != 3) $\wedge$ return==10
R2: (bufsize + 5 $>$ 0) $\wedge$ (bufsize + 5 + 10 $>$ 0) $\wedge$ (flags != 4) $\wedge$ (flags != 3) $\wedge$ return==10
R3: (bufsize + 5 $<=$ 0) $\wedge$ (bufsize + 5 + 10 $>$ 0) $\wedge$ (flags == 4) $\wedge$ (flags != 3) $\wedge$ return==0
R4: (bufsize + 5 $<=$ 0) $\wedge$ (bufsize + 5 + 10 $>$ 0) $\wedge$ (flags != 4) $\wedge$ (flags != 3) $\wedge$ return==0
π_{RN} = R1 $\vee$ R2 $\vee$ R3 $\vee$ R4
π'_{RN} = π_{RN} x IdMap$_{ref}$ = π_{RN}
Path constraints in Tar$_{new}$ covering ported edits and inconsistencies (π_{TN})
T1: (bufsize + 5 $>$ 0) $\wedge$ (bufsize + 5 + 10 $>$ 0) $\wedge$ return==10
T2: (bufsize + 5 $<=$ 0) $\wedge$ (bufsize + 5 + 10 $>$ 0) $\wedge$ return==0
π_{TN} = T1 $\vee$ T2
Path conditions in Ref$_{old}$ impacted by inconsistent nodes (π_{RO})
RO1: (bufsize + 5 + 10 $>$ 0) $\wedge$ (flags == 4) $\wedge$ (flags != 3) $\wedge$ return==0
RO2: (bufsize + 5 + 10 $>$ 0) $\wedge$ (flags != 4) $\wedge$ (flags != 3) $\wedge$ return==0
RO3: (bufsize + 5 + 10 $<=$ 0) $\wedge$ (flags == 4) $\wedge$ (flags != 3) $\wedge$ return==0
RO4: (bufsize + 5 + 10 $<=$ 0) $\wedge$ (flags != 4) $\wedge$ (flags != 3) $\wedge$ return==0
π_{RO} = RO1 $\vee$ RO2 $\vee$ RO3 $\vee$ RO4
π'_{RO} = π_{RO} x IdMap$_{ref}$ = π_{RO}
Path conditions in Tar$_{old}$ impacted by inconsistent nodes (π_{TO})
TO1: (bufsize + 5 + 10 $>$ 0) $\wedge$ return==10
TO2: (bufsize + 5 + 10 $<=$ 0) $\wedge$ return==0
π_{TO} = TO1 $\vee$ TO2

Step 3. Multi-Staged Equivalence Check

The program paths exercising inconsistent nodes and ported nodes from Step 2 (π_{RN} and π_{TN}) over-approximate the differential behavior because these paths are determined by a *may* analysis. Hence, in this step, using a multi-staged equivalence checker, we will compare the input-output characteristics of the reference paths (π_{RN}) with the target paths (π_{TN}) to check whether these paths indeed illustrate differential behavior. The final outcome will be a set of test cases showing the true differences. These test cases could help developers understand how a ported code snippet behaves differently in the target implementation w.r.t. the reference. Figure 6.4.2 shows the overview of our multi-staged equivalence checking algorithm. In the following subsections, we elaborate individual steps.

Check A: Are the program behaviors equivalent after ported edits?

We check here if the paths exercising inconsistent nodes and ported edits, π_{RN} and π_{TN}, are equivalent. To establish such equivalence, we first replace the reference identifiers present in π_{RN} with corresponding target identifiers using identifier map IdMap$_{ref}$, i.e., $\pi'_{RN} = Replace(\pi_{RN}, IdMap_{ref})$. We then check the behavioral equivalence according to Equation 6.2. If π'_{RN} is equivalent to π_{TN}, the ported code is behaving similarly in the reference and target contexts, and we exit without generating any test case.

$$CheckA : \pi_{TN} \equiv \pi'_{RN} \tag{6.2}$$

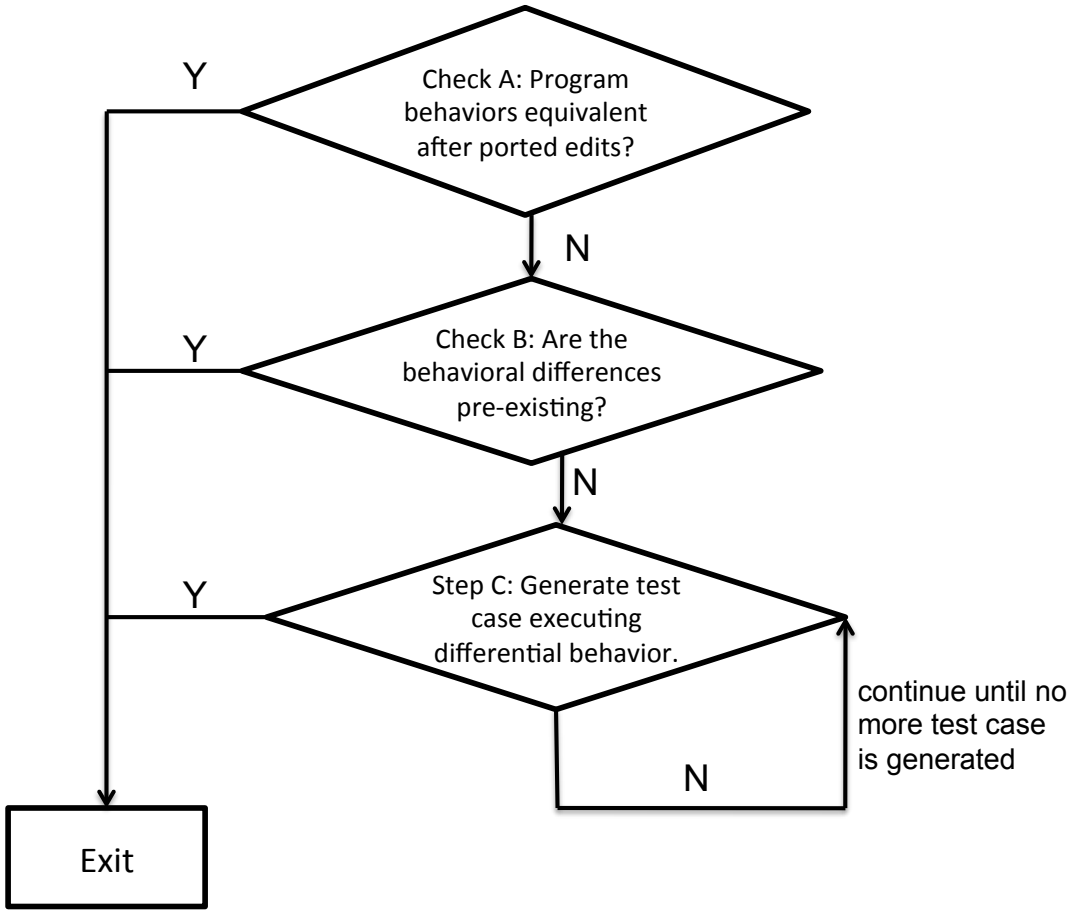


Figure 6.5: Multistaged Equivalence Checking

Check B: Are the behavioral differences caused by differing, pre-existing contexts?

Any semantic differences between π_{RN} and π_{TN} will cause $CheckA$ to fail. However, some of these differences may have pre-existed in the old program versions Tar_{old} and Ref_{old}. We do not consider such pre-existing differences as porting inconsistencies because they are not introduced by the ported code. Such pre-existing differences can be represented as $(\pi'_{RO} \wedge \pi_{TO})$, where π'_{RO} is generated by replacing all the reference identifiers by target identifiers using $IdMap_{ref}$, i.e., $\pi'_{RO} = Replace(\pi_{RO}, IdMap_{ref})$. $CheckB$ checks that if we disregard the pre-existing differences (i.e., $\neg(\pi'_{RO} \wedge \pi_{TO})$), whether the new versions could be equivalent. Equation 6.3 summarizes this process.

$$CheckB : \neg(\pi_{TO} \wedge \pi'_{RO}) \rightarrow (\pi_{TN} \equiv \pi'_{RN}) \tag{6.3}$$

Valid $CheckB$ means the differential behavior already existed in the original program versions, between Tar_{old} and Ref_{old} and ported edits do not introduce any new behavioral differences between Tar_{new} and Ref_{new}. In this case, we do not generate any test case.

Check C: Generate test cases that execute differential behaviors

If $CheckB$ is not valid, this may mean that the behavioral differences are coming from the interaction between ported edits and pre-existing contexts. To help developers explore such behavioral differences, we enumerate a new test case that does not satisfy $CheckB$ and that is not previously generated iteratively.

Suppose that `t1` is a solution of $CheckB$, which means `t1` is a test case that shows an inconsistent behavior arising from the interaction between a pre-existing context and ported edits. To generate the next test case, we check whether $CheckB$ can be satisfied when excluding `t1`. Similarly, we enumerate one test at a time until no new test case is generated.

$$\begin{aligned} C0 : (\neg t1) \rightarrow CheckB, \text{ where } t1 \text{ is a solution of } \neg\ CheckB \\ C1 : (\neg t2) \rightarrow C0, \text{ where } t2 \text{ is a solution of } \neg\ \text{C0} \\ .... \\ \text{(until no more new test case is generated)} \end{aligned} \tag{6.4}$$

Table 6.10: Results of Multi-Staged Equivalence Checking

Formula F to be checked	Valid(F)	Test Cases : Solve(Not(F))
$CheckA : \pi_{TN} \equiv \pi'_{RN}$	No	
$CheckB : (\neg\pi_{TO} \vee \neg\pi'_{RO}) \rightarrow CheckA$	No	t1: [bufsize = -5, flags = 3]
$C0 : (\neg t1) \rightarrow CheckB$	No	t2: [bufsize = 0, flags = 3]
$C1 : (\neg t2) \rightarrow C0$	Yes	

Table 6.10 illustrates the process of generating test cases for the motivating examples of Table 6.7. Path conditions illustrating inconsistent behaviors in the target and the reference programs (π_{TN}, π_{TO}, π_{RN}, and π_{RO}) are summarized in Table 6.9. The table also shows the transformed path conditions π'_{RN} and π'_{RO}, generated by replacing reference identifiers with target identifiers using IdMap$_{ref}$. In this example, $CheckA$ of the multi-staged equivalence checker fails, i.e., π_{TN}

$\not\equiv \pi'_{RN}$. That means, the differential behaviors due to ported edits in the reference and target contexts are indeed different. In this case, $CheckB$ also fails, showing the differences exist due to interaction between pre-existing differing contexts and the ported edits. Not($CheckB$) has solution `t1`. This means that with an input value of `bufsize = -5` and `flags = 3`, the behavior of Tar$_{new}$ and Ref$_{new}$ are different. The test case exercises path {t0, t1, t2, t3, t4, t6} in Tar$_{new}$ and path {r0, r1, r2} in Ref$_{new}$, which are clearly not equivalent. Next, we check $C0$, i.e., in absence of `t1` whether $CheckB$ is valid. $C0$ is also not valid. Not($C0$) has solution `t2` with an input value of `bufsize = 0` and `flags = 3`. `t2` corresponds to paths {t0, t1, t2, t3, t4, t5, t6} and {r0, r1, r2} in Tar$_{new}$ and Ref$_{new}$ respectively. Clearly, these two paths exhibit differential behavior. We next check if we ignore `t2` whether any other differences exist, i.e., we prove $C1$. $C1$ becomes valid. Here, we stop generating further test cases because no more differences exist. Thus, from the motivating example of Table 6.7 two concrete test cases `t1` and `t2` are generated that illustrate differential behaviors of the ported code in the target and reference implementations.

6.5 Summary

When porting code from one context to another, the semantics of the ported code often change due to differences in the surrounding contexts. Developers may overlook such subtle differences, inadvertently creating a *porting error*. Leveraging the categorization of porting errors from Chapter 5, we design SPA, an algorithm to detect and characterize semantic inconsistencies in ported code. Using

static control- and data flow analysis, SPA analyzes the semantics of ported edits in source and target contexts and identifies porting inconsistencies. Our evaluation of SPA on several large open-source code bases shows that SPA can detect porting inconsistencies with high precision and recall, and it outperforms two state-of-the-art techniques with 14% to 17% better precision.

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the contributions of the thesis in Section 7.1. We further discuss directions for future work in Section 7.2.

7.1 Summary

As a first step to measure the impact of cross-system porting on forked projects, we implement REPERTOIRE, a cross-system porting analysis tool. Given two program patches as input, REPERTOIRE identifies ported edits between them with 94% precision and 84% recall. REPERTOIRE also provides a visual interface to monitor cross-system porting. Using REPERTOIRE, managers and engineers can monitor how frequently developers port code, how long it usually takes to port patches from one project to another, which developers frequently participate in porting activities, and which subsystems are the sweet spot for porting. Thus, REPERTOIRE could help managers making more informed decisions on how to manage the co-evolution of a product family.

Using REPERTOIRE, we comprehensively characterize the temporal, spatial, and developer dimensions of cross-system porting in the BSD product family. We find that maintaining FreeBSD, NetBSD, and OpenBSD in parallel requires a

significant amount of porting work—on average 11% to 16% of the edits in each BSD release comes from porting. The cross-system porting activity is also periodic, and the porting rate does not necessarily decrease over time. Surprisingly, ported changes are less defect prone than non-ported changes. This may indicate developers usually port well-tested features and bug fixes. The efficiency of cross-system porting is heavily dependent on a small percentage of active developers doing their job on time.

To understand the extent and characteristics of porting errors in practice, we conduct a case study on porting errors using FreeBSD and Linux version histories. We find that developers often make mistakes while porting code and these errors take about a year to be detected and fixed on average. We also notice that porting errors are introduced due to some common adaptation mistakes including porting code to different control and data flow contexts, inconsistent renaming of identifiers, and redundancy.

Leveraging this categorization of porting errors, we design a static control- and data-dependence analysis technique, SPA, to detect and characterize porting inconsistencies. Our evaluation on ported code from four open-source projects shows that SPA can detect porting inconsistencies with 65% to 73% precision and 90% recall, and identify inconsistency types with 58% to 63% precision and 92% recall on average. SPA detects porting errors with 14% to 17% better precision than existing error detection tools.

This thesis is an effort to broaden the scope of co-evolution study to include entire project families as opposed to simply focusing on a single project. To

understand how similarly a project family co-evolves, we investigate cross-system porting in multiple dimensions. Together REPERTOIRE and SPA could serve software engineers to monitor co-evolution of similar systems and detect porting errors in the early phase of a development cycle.

7.2 Future Work

Extending SPA to detect copying errors. As part of our future work, we plan to integrate SPA with an integrated development environment so that developers can detect inconsistencies while copying code from a reference implementation and pasting it to a target implementation. For example, SPA can be deployed as a plugin extension of Eclipse. The plugin will track developer's copy-paste activities similar to Kim et al. [39] and use SPA to display the potential porting errors. To further assist developers reason about how the detected inconsistencies change the behavior of the ported code, we plan to generate test cases illustrating the differential behavior as sketched in Section 6.4.

Improving semantic porting analysis. Even though SPA outperforms existing tools in detecting porting errors, SPA's false positive is still high—27% to 30% on average. Such false positives may arise when two syntactically different statements are semantically same. For example, SPA mistakenly identifies a `for` statement and an equivalent `while` statement as inconsistent contexts. To reduce such false positives, we plan to compare the dynamic behaviors of ported code. Also, all the detected inconsistencies may not be errors. Since developers port code to different contexts, some of the porting inconsistencies may be intended. Based on this obser-

vation, we plan to investigate heuristics to rank the inconsistencies based on their error potential.

A notification system for collateral evolution. In a product family, when a related feature or bug fix is introduced in one project, it may need to be eventually ported to other projects. In Chapter 4, we show that the average time to port an edit from one system to another is more than 2 years. Sometimes it even takes up to 10 years for an edit to be propagated. We believe it will be helpful to build a notification system for co-evolution to promote a fast and timely patch application. When a relevant edit is introduced in one project, all the related components of other projects may need to be notified. However, we cannot notify all the changes, as the forked projects usually evolve independently in different directions. Naively notifying all the changes will significantly increase the false positives. First, we have to find which components across the projects are closely related in terms of cross-system porting. Since ported edits are not confined within files having similar names or directory structures, establishing such relationships can be challenging. Prior porting history may be useful for identifying relevant components. By extending auto patch recommendation systems like Sydit [52,53], Andersen et al. [11], we can also suggest and apply a patch automatically to relevant components.

Analysis of Backporting. Our approach to determine repetitive work is not limited to only forked projects. Any systems that maintain multiple parallel versions can benefit from REPERTOIRE. For example, Linux has a mainline branch where developers commit their recent developments. Linux also maintains some stable branches to provide long term support to its older releases. Sometimes it becomes

important to push critical features or bug fixes from mainline to the stable branches. This is called backporting [59]. Using REPERTOIRE, we can study the extent and characteristics of backporting between different versions of a project. Especially, we would like to investigate (1) What percentage of mainline commits are backported? (2) What are the characteristics of backported patches—bugfixes, feature additions, new functionalities, etc.? (3) How different is a backported patch with respect to its original main-line patch?, and (4) How much time does it take to test a backported patch? We believe these questions could help us to understand the effort of maintaining parallel versions of a project.

Studying bug report similarities in a product family. We would like to investigate whether similar bug reports in two forked projects are fixed similarly. Extending prior approaches to find duplicate bug reports [56, 72, 78], we can find similar bug reports across different forked projects and extract their corresponding fixed patches. Then, REPERTOIRE can verify how similar these fixed patches are. This will help us to determine whether two similar bug reports in two related projects undergo similar fixes. If we confirm that a large number of fixes are actually similar, it would motivate building a patch recommendation system. When a new bug will be reported in a project, we will look for a similar bug report in the sibling projects and suggest its fix.

Appendices

Appendix A

Repertoire

REPERTOIRE is an open source tool and can be downloaded from `http://dolphin.ece.utexas.edu/Repertoire.html` This section describes the steps required to run REPERTOIRE.

A.1 Installation

1. Install required libraries
   - `Python 2.7`
   - `Qt 4.x`: a cross-platform application and UI framework
   - pyuic4: a UI compiler for `Qt` that comes with the `PyQT` package.
2. Run `'make'` from `src/`
3. Run `'make'` from `src/analysis/`
4. Obtain a working copy of CCFinderX for your platform
   - Ensure the execution of CCFinderX by running a command `'ccfx d cpp somefile.cpp'`

A.2 Populating a Database

REPERTOIRE takes as input the repository location and time period of version histories and identifies ported edits among the input projects. It requires a working directory to store intermediate files, a path to an executable CCFinderX, and information about version control repositories. For each repository, the user is asked to specify the type of version control system (e.g. Git or Mercurial), the root URL of the repository, and the time period that the user is interested in. Table 3.5 shows example inputs. REPERTOIRE checks the validity of inputs and then proceeds to populate a database with the analysis results of ported edits.

1. Run `'src/run_vcs_flow.py'`
   - When an input wizard appears, select "Start a new project"
2. Pick a working directory, e.g. `/var/tmp`.
   - Repertoire creates a sub directory to put its intermediate results.
3. Specify a path to a CCFinder executable.
   - You may optionally pick a minimum token size (CCFinder's input parameter). A minimum token size is the number of lexical token elements that must be similar between two code fragments to be identified as code clones.
4. Select a version control system for each project.

- REPERTOIRE currently supports Git or Mercurial as target version control systems. Alternatively, a user may provide a directory including pre-extracted *diff*-based patches.

5. Select a URL path for each version control repository.
   - This is the root directory of the repository for Git and Mercurial.
6. Select file extensions for C/C++, headers, and Java files. This step allows users to ignore ported edits in certain types of files.
7. Select a time period for the project. Repertoire then extracts *diff-based* patches for each commit revision within the time period.
8. Confirm analysis of the given data and then wait for analysis to complete.
9. When the analysis is complete, check the output file created by REPERTOIRE in the working directory.
   - There will be a `pickle` file called `rep_db.pickle`, which is a file format for Python object serialization and de-serialization. This is used as an input for the visualization and analysis step.

A.3 Running Repertoire

1. Run `rep_analysis.py` from `src/analysis`
2. Select the pickle file `rep_db.pickle` produced from the previous step and press `Next`.

3. The GUI provides four analysis views shown in Figure A.1: Porting Frequency View, File Distribution View, Developer View, and Porting Latency View (Timing Analysis).

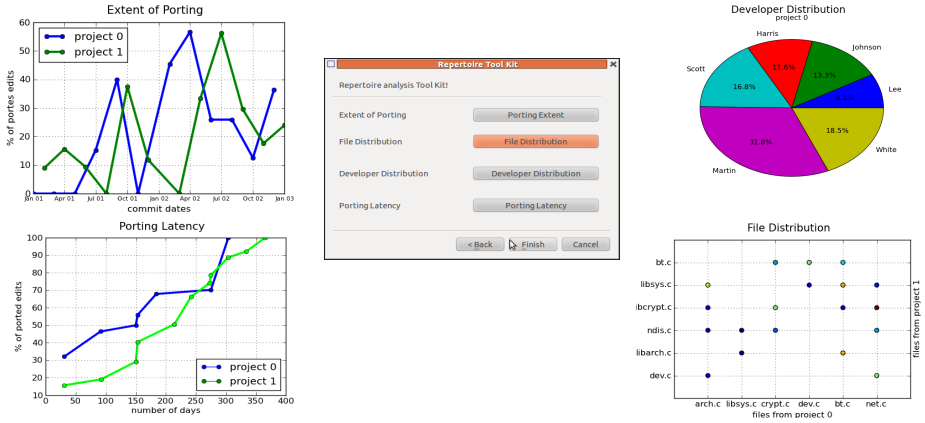


Figure A.1: REPERTOIRE Analysis Menu

A.4 Porting Frequency View

Given the version histories of two projects, this view shows the extent of edits ported from one project to another over the available history. This is represented as a line diagram, where x-axis shows a time line, and the y-axis shows the average percentage of ported edits with respect to total edits in *diff*-based patches. A user may select to see only ported edits from Project A to B, B to A, or both ways at once. Figure 3.2 shows an example of this porting frequency view. Steps to run this view:

1. Select Porting Frequency in the menu.

2. Select a project: Project 0 orand Project 1

3. Set a time period for analysis.

A.5 File Distribution View

This view is a scatter plot where files from Project A is shown along the x-axis and files from Project B is shown along the y-axis. A point is plotted at `(x,y)` if there is an edit ported from file X to file Y or vice versa. The color of the dot indicates a ratio of ported edits to total edits. The darker the color is, the higher density of ported edits. Figure 3.3 shows an example. Steps to run this view:

1. Select File Distribution in the menu.

2. By default, this view does not show full file names. A user can click Display Label option to see the full file names.

3. When a user click on the point in the diagram, corresponding files names are shown at the bottom.

4. To browse ported code between the selected file pair, press Display Ported Edit

5. A window will show all ported code fragments between the two files, along with developer and commit date information.

6. On selecting any clone from clone list, user can browse the ported edit. Figure 3.4 shows an example these last two steps.

A.6 Developer View

Figure 3.5 shows an example of developer distribution. The pie chart shows which developers are responsible for what fraction of the total ported lines. The scatter diagram in this figure with developers of project 0 in x-axis and developers of project 1 in y-axis also reflects the interaction pattern of the developers while porting code. Steps to run the developer analysis:

1. Select Developer Distribution in the menu.
2. Shows a scatter plot of developer distribution, i.e., a point is plotted at (x,y) if developers at x port code written by developer at y, and vice versa.
3. We do not show developers names as label initially, as it clutters the display. User can see labels however, if Display Label is pressed.
4. If any point on the diagram is pressed, corresponding developer names can be seen at the bottom.
5. To see developer's distribution in a particular project in the form of pie chart, please select project 0 and/or project 1 from right hand window. Then press Display Developer Porting Statistics"

A.7 Porting Latency View

This analysis shows how long it takes for a patch to be propagated to the other project on average. Figure 3.6 an example of this view. Steps to run this analysis:

1. Select Porting Latency in the menu.

2. Select a project (e.g. Project A or B), and then press Porting Latency button.

3. A cumulative distribution of the time taken to port edits from the source to target projects is shown when pressing Cumulative Distribution.

4. A user may limit the time period to a specific time period for an in-depth investigation.

Appendix B

Porting Error

Table B.1: The data set of porting errors in FreeBSD identified through the log message analysis

	File Name	Commit Logs
1	src/sys/dev/et/if_et.c	date: 2009/11/19 22:59:52; author: yongari; state: Exp; lines: +2 -2 SVN rev 199563 on 2009-11-19 22:59:52Z by yongari Fix *copy & paste error* and remove extra space before colon.
2	src/sys/netinet6/ip6_forward.c	date: 2012/05/25 02:17:16; author: bz; state: Exp; lines: +0 -2 SVN rev 235958 on 2012-05-25 02:17:16Z by bz MFp4 bz_ipv6_fast: Add support for delayed checksum calculations in the IPv6 output path. We currently cannot offload to the card if we add extension headers (which incl. fragmentation). Fix two SCTP offload support *copy&paste bugs*: calculate checksums if fragmenting and no need to flag IPv4 header checksums in the IPv6 forwarding path.
3	src/lib/libc/locale/toupper.c	date: 2012/05/10 20:03:34; author: dim; state: Exp; lines: +1 -1 SVN rev 235239 on 2012-05-10 20:03:34Z by dim Fix *copy/paste error* in lib/libc/locale/toupper.c.

4	src/sys/dev/e1000/if_igb.h	date: 2012/04/25 02:05:14; author: emaste; state: Exp; lines: +1 -1 SVN rev 234665 on 2012-04-25 02:05:14Z by emaste Fix *cut-and-paste* comment *error*
5	src/sys/powerpc/-include/profile.h	date: 2012/04/23 20:53:50; author: nwhitehorn; state: Exp; lines: +1 -1 SVN rev 234615 on 2012-04-23 20:53:50Z by nwhitehorn Fix *copy-and-paste error* in r230400. MFC after: 3 days revision 1.10.2.3 date: 2012/04/26 14:02:39; author: nwhitehorn; state: Exp; lines: +1 -1 branches: 1.10.2.3.2; SVN rev 234708 on 2012-04-26 14:02:39Z by nwhitehorn MFC r234615: Fix *copy-and-paste error* in r230400 that would cause ppc64 executables built with -fvisibility=hidden to fail to link with a message about hidden symbol main being referenced from a DSO.
6	src/lib/libthr/thread/thr_umtx.c	date: 2012/02/19 08:17:14; author: davidxu; state: Exp; lines: +1 -1 SVN rev 231906 on 2012-02-19 08:17:14Z by davidxu Check both seconds and nanoseconds are zero, only checking nanoseconds is zero may trigger timeout too early. It seems a *copy&paste bug*.
7	src/sys/dev/qlxgb/qla_misc.c	date: 2012/01/03 20:51:26; author: dim; state: Exp; lines: +1 -1 SVN rev 229423 on 2012-01-03 20:51:26Z by dim In sys/dev/qlxgb/qla_misc.c, fix a *copy/paste issue*. Clang complained the variable ‘val’ was uninitialized when used. Instead, ‘sig’ should have been printed.

8	src/contrib/ntp/ntpd/ntp_io.c	date: 2011/05/29 07:40:48; author: bz; state: Exp; lines: +7 -2 SVN rev 222444 on 2011-05-29 07:40:48Z by bz ... While here also fix the *copy&paste error* in the log message for IPV6_MULTICAST_LOOP. Reviewed by: roberto Sponsored by: The FreeBSD Foundation Sponsored by: iXsystems MFC after: 10 days Filed as: Bug 1936 on ntp.org
9	src/sys/kern/vfs_syscalls.c	Done: indentation Working file: revision 1.505 date: 2011/04/18 16:40:47; author: mdf; state: Exp; lines: +3 -3 SVN rev 220793 on 2011-04-18 16:40:47Z by mdf Fix a *copy/paste* whitespace *error*.
10	src/sys/net80211/ieee80211-sta.c	date: 2011/03/13 12:21:04; author: bschmidt; state: Exp; lines: +1 -1 SVN rev 219603 on 2011-03-13 12:21:04Z by bschmidt Fix a *cut&paste error* while parsing htcap/ht-info elements. This one is reponsible for not filling ni_htrates if a pre-ht information element is present.
11	src/sys/dev/e1000/if_em.c	date: 2010/12/04 01:59:58; author: jfv; state: Exp; lines: +32 -36 SVN rev 216172 on 2010-12-04 01:59:58Z by jfv Small *cut and paste bug* in flow control string fixed.
12	src/lib/libusb/libusb10.c	date: 2010/10/14 20:50:33; author: hselasky; state: Exp; lines: +72 -12 SVN rev 213853 on 2010-10-14 20:50:33Z by hselasky - Correct some wrong error codes due to *copy and paste error*.

13	src/sys/crypto/aesni/aesni_wrap.c	date: 2010/09/25 10:32:52; author: pjd; state: Exp; lines: +2 -2 SVN rev 213166 on 2010-09-25 10:32:52Z by pjd Fix two *copy&paste bugs*.
14	src/lib/libthr/thread/thr_private.h	date: 2010/09/13 11:57:46; author: davidxu; state: Exp; lines: +2 -2 SVN rev 212551 on 2010-09-13 11:57:46Z by davidxu Fix *copy&paste problem*.
15	src/sys/dev/usb/wlan/if_rum.c	date: 2010/06/14 23:01:50; author: jkim; state: Exp; lines: +1 -1 SVN rev 209189 on 2010-06-14 23:01:50Z by jkim Fix typos that broke duration calculations on protection frames. A similar fix was done for ral(4) long ago and it must be *copy-and-paste bugs*.
16	src/sys/cam/ata/ata_xpt.c	revision 1.16 date: 2009/11/25 14:24:14; author: mav; state: Exp; lines: +1 -1 SVN rev 199799 on 2009-11-25 14:24:14Z by mav Fix small *copu-paste bug*.
17	src/lib/libc/gen/fmtmsg.c	date: 2009/11/08 14:02:54; author: brueffer; state: Exp; lines: +1 -1 branches: 1.6.2; SVN rev 199046 on 2009-11-08 14:02:54Z by brueffer Fix a *copy+paste error* by checking the correct variable against MM_NULLACT.
18	src/lib/libkvm/kvm_i386.c	date: 2009/11/06 13:10:12; author: jhb; state: Exp; lines: +6 -6 SVN rev 198986 on 2009-11-06 13:10:12Z by jhb Fix a *copy-paste bug* when reading data from the last 3 (7 for PAE) bytes of a page mapped by a large page in the kernel.

19	src/lib/libkvm/kvm_i386.c	date: 2009/11/06 13:10:12; author: jhb; state: Exp; lines: +6 -6 SVN rev 198986 on 2009-11-06 13:10:12Z by jhb Fix a *copy-paste bug* when reading data from the last 3 (7 for PAE) bytes of a page mapped by a large page in the kernel.
20	src/sys/dev/ppbus/lpt.c	date: 2009/10/13 12:23:28; author: jhb; state: Exp; lines: +1 -1 SVN rev 198028 on 2009-10-13 12:23:28Z by jhb Correct a *copy/paste bug* in a comment. lpt-close() checks once a second to see if the ppc hardware has gone idle rather than four times a second.
21	src/sys/dev/ixgbe/ixgbe.c	date: 2009/09/03 22:00:42; author: jfv; state: Exp; lines: +1 -1 SVN rev 196798 on 2009-09-03 22:00:42Z by jfv Stupid *cut and paste* error on a stats struct member, thanks to Ryan at Small Tree for finding this one.
22	src/sys/cam/ata/ata_xpt.c	date: 2009/08/18 09:27:17; author: mav; state: Exp; lines: +4 -4 SVN rev 196353 on 2009-08-18 09:27:17Z by mav Fix *copy/paste bug*, that requests data read during ATA device probe sequence for ATA_SETFEATURES/ATA_SF_SETXFER command which by definition transfers no data. Most of controllers are irrelevant to this bug, but some nVidia's doesn't. Tested on: current@ Approved by: re (kib)

23	rc/sys/netinet/ipfw/ip_fw2.c	date: 2009/08/14 10:09:45; author: julian; state: Exp; lines: +1 -1 SVN rev 196201 on 2009-08-14 10:09:45Z by julian Fix ipfw crash on uid or gid check. Receiving any ip packet for which there is no existing socket will crash if ipfw has a uid or gid test rule, as the uid/gid of the non existent owner of said non existent socket is tested. Brooks introduced this error as part of his >16 gids patch. It appears to be a *cut-n-paste error* from similar code a few lines before. The old code used the 'pcb' variable here, but in the new code that switched the 'inp' variable, which is often NULL and what is tested in the code further up. The rest of the multi-gid patch for ipfw seems solid (and cleaner than previous code).
24	src/sys/cam/ata/ata_da.c	date: 2009/07/17 21:48:08; author: mav; state: Exp; lines: +3 -2 branches: 1.2.2; SVN rev 195748 on 2009-07-17 21:48:08Z by mav Fix *copy-paste bug*. Use regular non-polled mode for executing FLUSHCACHE command on disk close. Approved by: re (implicitly)
25	src/sys/cam/ata/ata_xpt.c	date: 2009/07/13 21:21:30; author: mav; state: Exp; lines: +1 -1 branches: 1.3.2; SVN rev 195665 on 2009-07-13 21:21:30Z by mav Fix *copy-paste bug*, enabling SIM PMP support, when it was not really found.
26	src/sys/netgraph/netflow /ng_netflow.c	date: 2009/05/13 02:26:34; author: mav; state: Exp; lines: +1 -1 branches: 1.20.2; SVN rev 192032 on 2009-05-13 02:26:34Z by mav Fix *copy-paste bug* in NGM_NETFLOW_SETCONFIG argument size verification.

27	src/sys/dev/sound/pci/hda-/hdac.c	date: 2009/02/27 23:49:26; author: mav; state: Exp; lines: +1 -1 SVN rev 189127 on 2009-02-27 23:49:26Z by mav *Copy/paste bug* fix for previos commit.
28	src/sys/dev/mxge/if_mxge.c	date: 2009/02/17 22:25:19; author: gallatin; state: Exp; lines: +1 -1 SVN rev 188737 on 2009-02-17 22:25:19Z by gallatin Fix *cut/paste error* in previous commit and use the correct value for SFP+ reserved media type.
29	src/sys/dev/usb2/serial-/uslcom2.c	ate: 2009/02/15 23:38:58; author: thompsa; state: Exp; lines: +7 -7 SVN rev 188664 on 2009-02-15 23:38:58Z by thompsa Make uslcom compile, *cut'n'paste errors* from uplcom.
30	src/sys/dev/usb2/serial-/uslcom2.c	ate: 2009/02/15 23:38:58; author: thompsa; state: Exp; lines: +7 -7 SVN rev 188664 on 2009-02-15 23:38:58Z by thompsa Make uslcom compile, *cut'n'paste errors* from uplcom.
31	src/sys/dev/e1000/if_igb.c	date: 2009/01/13 00:10:50; author: gnn; state: Exp; lines: +1 -1 SVN rev 187123 on 2009-01-13 00:10:50Z by gnn Fix a *cut/paste bug* which prevents us from setting the average latency tunable.
32	src/sys/dev/usb/u3g.c	date: 2008/11/03 22:09:27; author: n_hibma; state: Exp; lines: +22 -18 SVN rev 184605 on 2008-11-03 22:09:27Z by n_hibma Bugfix: *Cut&paste error* from the NetBSD code.

33	src/sys/i386/include-/pmc_mdep.h	date: 2008/09/05 14:45:56; author: jkoshy; state: Exp; lines: +2 -2 SVN rev 182790 on 2008-09-05 14:45:56Z by jkoshy Correct a *copy-paste error*—do not look for REX prefixes in i386 code.
34	src/sys/dev/mii/rgephy.c	date: 2008/08/06 07:52:59; author: kevlo; state: Exp; lines: +0 -1 branches: 1.21.2; SVN rev 181343 on 2008-08-06 07:52:59Z by kevlo Fix a *copy/paste error*
35	src/sys/netinet/libalias/alias.c	date: 2008/06/01 11:47:04; author: mav; state: Exp; lines: +2 -2 SVN rev 179472 on 2008-06-01 11:47:04Z by mav Fix packet fragmentation support broken by *copy/paste error* in rev.1.60. ip_id should be u_short, but not u_char.
36	src/sbin/atacontrol/atacontrol.c	date: 2008/03/16 17:54:55; author: phk; state: Exp; lines: +67 -104 *Un-cut&paste* argument processing, fix things lint found.
37	src/sys/net/if.h	date: 2007/12/15 22:06:23; author: kmacy; state: Exp; lines: +1 -1 fix bonehead *cut and paste error* in last commit
38	src/usr.sbin/dconschat-/dconschat.c	date: 2007/07/12 13:08:00; author: simokawa; state: Exp; lines: +4 -1 branches: 1.15.2; 1.15.6; 1.15.8; 1.15.10; 1.15.12; Set the default escape character as described in the manpage of dconschat(8). Fix a *cut-and-paste error*.
39	src/sys/dev/mxge/if_mxge.c	date: 2007/04/27 13:11:50; author: gallatin; state: Exp; lines: +35 -19 -Fix an mbuf leak caused by a *cut&paste bug* where the small ring's mbufs were never freed, but the big ring was freed twice.

40	src/sys/netinet/sctputil.c	date: 2007/04/23 00:51:49; author: rrs; state: Exp; lines: +6 -6 Fixes *cut and paste bug* using wrong pointer reference.
41	src/sys/arm/xscale/-ixp425/if_npe.c	date: 2007/02/10 15:43:58; author: mlaier; state: Exp; lines: +1 -1 branches: 1.5.2; Fix small altq related *copy and paste error*.
42	src/sys/arm/at91/if_ate.c	date: 2007/02/10 15:43:57; author: mlaier; state: Exp; lines: +1 -1 Fix small altq related *copy and paste error*.
43	src/tools/regression/-lib/libc/stdio/test-scanfloat.c	date: 2007/01/03 05:38:08; author: das; state: Exp; lines: +2 -2 Fix *cut-and-paste bugs* in the regression tests.
44	src/sys/kern/sched_4bsd.c	date: 2006/11/14 05:48:27; author: davidxu; state: Exp; lines: +12 -12 Fix a *copy-paste bug* in NON-KSE case.
45	src/sys/dev/agp/agp_i810.c	date: 2006/09/27 06:38:54; author: anholt; state: Exp; lines: +29 -13 Add support for 945G/GM AGP chipsets. Also corrected is a minor copy-and-pasteo in an error case.
46	src/sys/pci/agp_i810.c	Done: Working file: revision 1.39 date: 2006/09/27 06:38:54; author: anholt; state: Exp; lines: +29 -13 Add support for 945G/GM AGP chipsets. Also corrected is a minor copy-and-pasteo in an error case.

47	src/sbin/gpt/gpt.c	date: 2006/07/07 02:44:23; author: marcel; state: Exp; lines: +6 -6 branches: 1.16.2; Fix *cut-n-paste bug*: compare argument s against known aliases, not the global optarg. This bug goes unnoticed because optarg is so far always the actual argument for the formal argument s.
48	src/sys/compat/linux/-linux_stats.c	date: 2006/05/05 16:17:59; author: ambrisko; state: Exp; lines: +0 -1 Fix the the duplicate *cut-n-paste* in linux_fstat64 pointed out by Alexander Leidinger. I forget to fix it in this version.
49	src/sys/dev/hwpmc/-hwpmc_mod.c	date: 2006/04/11 01:15:26; author: jkoshy; state: Exp; lines: +2 -2 Fix a *cut-n-paste* bug that crept in. Reported by: ”Pawel Worach” pawel.worach at gmail.com
50	src/sys/netgraph/ng_base.c	date: 2005/07/21 22:15:37; author: glebius; state: Exp; lines: +1 -1 Fix *cut-n-paste* error, introduced in rev. 1.103.
51	src/sys/alpha/osf1/-osf1_mount.c	date: 2005/06/11 11:46:32; author: pjd; state: Exp; lines: +1 -1 Fix *copy&paste bug*.
52	src/sys/dev/vkbd/vkbd.c	date: 2005/05/20 23:29:55; author: emax; state: Exp; lines: +1 -1 branches: 1.7.2; Fix yet another *cut-and-paste bug*. kbd was allocated from M_VKBD not from M_DEVBUF

53	src/sys/netgraph/bluetooth-/socket/ng_btsocket.c	date: 2005/04/06 20:54:05; author: emax; state: Exp; lines: +1 -1 branches: 1.11.2; Remove PR_ATOMIC flag in ng_btsocket_protosw[] for BLUETOOTH_PROTO_RFCOMM protocol. RFCOMM is a SOCK_STREAM protocol not SOCK_SEQPACKET. This was a serious bug caused by *cut-and-paste*. I'm surprised it did not bite me before. Dunce hat goes to me.
54	src/sys/dev/ixgb/if_ixgb.c	date: 2005/03/27 17:22:41; author: mux; state: Exp; lines: +2 -2 Fix *copy&paste error* in my previous commit. Spotted by: ru
55	src/sys/dev/pci/pcireg.h	date: 2005/03/26 22:17:48; author: jmg; state: Exp; lines: +2 -2 fix a *copy/paste typo* for scanner/gameport...
56	src/sys/arm/arm/busdma-_machdep.c	date: 2005/03/08 11:18:14; author: mux; state: Exp; lines: +21 -23 Use __func__ in the KTR_BUSDMA traces. This avoids *copy and paste errors* like in the bus_dmamap_load_mbuf_sg() case where we were wrongly displaying the function name as bus_dmamap_load_mbuf.
57	src/sys/dev/bktr/bktr_audio.c	date: 2005/03/02 10:27:35; author: obrien; state: Exp; lines: +3 -3 MFC: + Remove lots of tab/space errors introduced by massive *cut-n-paste* action.

58	src/sys/dev/bktr/bktr_tuner.h	date: 2005/03/02 10:27:35; author: obrien; state: Exp; lines: +1 -1 MFC: + Remove lots of tab/space *errors* introduced by massive *cut-n-paste* action. + Take into account that Pinnacle screwed up their PCI ID in the beginning.. Older cards have it reversed. Also, use some already defined values instead of magic numbers.+ Add code to do better auto detection of tuner types etc.+ Add support for the Pixelview PlayTV + Remove vnode.h and adjust includes to compensate for pollution. + Remove support for FreeBSD ¡ 4.recent from this driver.
59	src/sys/geom/stripe/g_stripe.h	date: 2004/07/18 16:51:58; author: pjd; state: Exp; lines: +1 -1 Fix *copy&paste bug*.
60	src/sys/net/if_var.h	date: 2004/07/14 13:31:41; author: mlaier; state: Exp; lines: +5 -5 Fix a *copy-and-paste-o* in IFQ_DRV_PREPEND - all pointyhats to me. While here also fix a (not less stupid) braino in IFQ_DRV_PURGE. Reported-by: clement Tested-by: clement (_PREPEND in sis(4))
61	src/usr.bin/tar/write.c	date: 2004/05/03 16:56:42; author: kientzle; state: Exp; lines: +4 -3 Correct *copy/paste error* in Linux nodump support. Thanks to: Juergen Lock for his continuing patience while I botch his patches.
62	src/sys/contrib/pf/net/pf.c	date: 2004/04/11 17:35:40; author: mlaier; state: Exp; lines: +11 -6 Commit import of OpenBSD-stable fix: 1.432 Fix icmp checksum when sequence number modlation is being used. Also fix a daddr vs saddr *cut-n-paste error* in ICMP error handling.

63	src/sys/netgraph/ng_l2tp.c	date: 2004/04/04 21:33:09; author: archie; state: Exp; lines: +3 -3 Rename internal structure to fix *cut & paste error*.
64	src/lib/libatm/ip_addr.c	date: 2003/07/29 13:51:53; author: harti; state: Exp; lines: +1 -1 Correct a *cut'n'paste error* in a comment.
65	src/lib/libkse/thread/thr_info.c	date: 2003/07/07 12:12:33; author: davidxu; state: Exp; lines: +20 -11 Correctly print signal mask, the bug was introduced by *cut and paste* in last commit.
66	src/lib/libpthread/-thread/thr_info.c	date: 2003/07/07 12:12:33; author: davidxu; state: Exp; lines: +20 -11 Correctly print signal mask, the bug was introduced by *cut and paste* in last commit.
67	src/lib/libc/ia64-/gen/makecontext.c	date: 2003/06/02 00:16:39; author: marcel; state: Exp; lines: +2 -2 o Fix a *cut-n-paste bug*. We were clobbering rp with gp... o Make sure the arguments to ctx_wrapper() are loaded from the backing store by forcing an underflow. Do this by making all registers in the register frame local.
68	src/sys/ia64/ia64/trap.c	date: 2003/05/29 05:09:15; author: marcel; state: Exp; lines: +1 -1 Fix what I think is a *cut-n-paste bug*: use OID_AUTO for the print_usertrap sysctl instead of CPU_UNALIGNED_PRINT. The latter is used already. Approved by: re@ (blanket)

69	src/sys/netgraph/ng_bridge.c	Done: Working file: revision 1.19 date: 2003/05/15 18:51:28; author: julian; state: Exp; lines: +1 -1 fix a *cut-n-paste error*. in the case where the bridge node was closed down but a timeout still applied to it, the final reference to the node was freeing the private data structure using the wrong malloc type.
70	src/lib/libufs/block.c	date: 2003/03/30 18:00:24; author: jmallett; state: Exp; lines: +1 -1 MFp4: Fix *copy&paste* English *error*.
71	src/lib/libkse/thread/thr-_attr_get_np.c	date: 2003/03/05 20:50:03; author: phantom; state: Exp; lines: +1 -1 Fix *cut'n'paste error*
72	src/lib/libpthread/thread/thr-_attr_get_np.c	date: 2003/03/05 20:50:03; author: phantom; state: Exp; lines: +1 -1 Fix *cut'n'paste error*
73	src/sys/sparc64/sbus/sbus.c	revision 1.7 date: 2003/01/06 16:36:05; author: tmm; state: Exp; lines: +36 -14 1.) fix a *copy-and-paste-o* in a panic() message
74	src/sys/kern/vfs_bio.c	date: 2003/01/05 22:01:08; author: phk; state: Exp; lines: +2 -2 Fix *cut&paste bug* which would result in a panic because buffer was being biodone'ed multiple times.
75	src/sys/security/mac-_mls/mac_mls.c	date: 2002/10/22 18:36:47; author: rwatson; state: Exp; lines: +1 -1 s/mls/biba/ in a *copy+paste error* for a printf
76	src/sys/sys/kse.h	date: 2002/10/01 17:47:44; author: archie; state: Exp; lines: +1 -1 Fix *cut&paste error*: "tm_spare" should have been "km_spare". Noticed by: ru

77	src/sys/compat/pecoff/imgact_pecoff.c	date: 2002/09/07 07:13:08; author: peter; state: Exp; lines: +1 -0 Fix a missing line in a *cut/paste error*.
78	src/sys/dev/dc/if_dc.c	date: 2002/08/23 23:49:02; author: alfred; state: Exp; lines: +116 -58 style: put return values on a line by themselves. fix some *paste issues* where whitespace was used instead of tabs.
79	src/sys/dev/sf/if_sf.c	date: 2002/08/23 23:49:02; author: alfred; state: Exp; lines: +64 -32 style: put return values on a line by themselves. fix some paste issues where whitespace was used instead of tabs.
80	src/sys/dev/sk/if_sk.c	date: 2002/08/23 23:49:02; author: alfred; state: Exp; lines: +91 -47 style: put return values on a line by themselves. fix some *paste issues* where whitespace was used instead of tabs.
81	src/sys/netinet/ip_input.c	date: 2002/06/23 09:15:43; author: luigi; state: Exp; lines: +6 -7 fix *bad indentation* and whitespace resulting from *cut&paste*
82	src/sys/kern/kern_mutex.c	date: 2002/05/21 21:27:05; author: jhb; state: Exp; lines: +1 -1 Fix an old *cut 'n' paste bug* inherited from BSD/OS: don't increment 'i' twice once we are in the long wait stage of spinning on a spin mutex.
83	src/sys/alpha/pci/irongate_pci.c	date: 2002/05/10 16:56:14; author: gallatin; state: Exp; lines: +2 -2 Remove ## concatination in the CFGREAD and CFGWRITE macros, as gcc3 complains about them & they are not needed. Same fix as to tsunami_pci.c. (not surprising, as this code was cut and pasted from there when I wrote it).

84	src/lib/libdevstat/devstat.c	date: 2001/08/21 05:23:37; author: ken; state: Exp; lines: +37 -35 Fix some style inconsistencies introduced in rev 1.10, as well as some other inconsistencies that I missed in my review of rev 1.7. Also fix a *cut-n-paste error* from an earlier revision.
85	src/usr.sbin/config/config.h	date: 2001/02/28 02:53:32; author: peter; state: Exp; lines: +6 -8 Some more tidying up. we dont use config-dependent anyware. Eliminate some duplicate code (*cut/paste bug*?). tidy up some other minor stuff.
86	src/sys/netgraph/ng_bpf.c	date: 2001/01/30 07:58:30; author: julian; state: Exp; lines: +2 -2 Fix *cut and paste error* in a comment. Submitted by: Peter Wemm ¡peter@freebsd.org¿
87	src/sys/dev/ed/if_ed_pccard.c	date: 2000/11/25 07:25:08; author: peter; state: Exp; lines: +4 -4 Argh, I have fixed this *cut/-paste error* twice before. I must have committed the wrong patch. :-(sn_pccard_products[] should have been static anyway.
88	src/sys/dev/sound/isa/sb16.c	date: 2000/11/06 02:47:43; author: cg; state: Exp; lines: +3 -2 fix *paste*-o in mixer code - actually set right channel volume instead of doing the left channel twice.
89	src/sys/dev/usb/uscanner.c	date: 2000/11/01 00:28:40; author: n_hibma; state: Exp; lines: +2 -2 *Cut&paste bug*: Set USBD_SHORT_XFER_OK unconditionally

90	src/usr.bin/telnet/telnet.c	date: 2000/09/20 23:07:04; author: imp; state: Exp; lines: +7 -4 Fix buffer overflow when DISPLAY is longer than 43 characters. This is not exploitable because telnet doesn't run with elevated privs. Didn't fix all the other potential buffer overflows. Would be a good task for someone who has lots of time to carefully study each case because *cut and paste* solutions are dangerous for this code base. Added $FreeBSD$ in the same way that command.c did it.
91	src/sbin/ipfw/ipfw.c	date: 2000/07/17 03:02:15; author: billf; state: Exp; lines: +2 -2 Fix a *paste*-o in the tcpoptions check (not a security problem, just a error in the usage printf())
92	src/usr.bin/ftp/main.c	date: 2000/06/20 15:36:38; author: se; state: Exp; lines: +3 -3 Fix obvious *cut-n-paste error*. Submitted by: Thomas Ludwig ¡tludwig@urbanet.ch¿
93	src/lib/libkvm/kvm_file.c	date: 2000/02/18 16:39:00; author: peter; state: Exp; lines: +2 -2 branches: 1.11.2; Correct an *error* message presumably as a result of *cut/paste*. kvm_getfiles() referred to itself as kvm_getprocs().
94	src/lib/libtacplus/taclib.c	date: 2000/01/17 04:26:09; author: jdp; state: Exp; lines: +2 -2 branches: 1.2.2; Fix error message that was too hastily cut&pasted from libradius.
95	src/sys/netgraph/ng_socket.c	Working file: revision 1.9 date: 1999/11/21 23:11:52; author: julian; state: Exp; lines: +2 -2 oops *cut-n-paste error*

96	src/sys/dev/buslogic/bt_isa.c	date: 1999/04/18 19:08:28; author: peter; state: Exp; lines: +11 -5 Make the bt isa driver work.. - fix *cut/paste problem*. :-)
97	src/sys/dev/buslogic/bt_isa.c	date: 1999/04/18 19:08:28; author: peter; state: Exp; lines: +11 -5 Make the bt isa driver work.. - *fix cut/paste problem*. :-)
98	src/lib/libc/stdlib/realpath.c	date: 1999/02/12 19:45:53; author: ache; state: Exp; lines: +5 -5 fix tabs lost apparently in *copy&paste*
99	src/libexec/rtld-elf/alpha/reloc.c	date: 1998/09/08 09:47:35; author: dfr; state: Exp; lines: +2 -2 Fix a *cut&paste error* which prevented LD_BIND_NOW from working.
100	src/sys/net/rtsock.c	date: 1997/07/16 14:55:14; author: julian; state: Exp; lines: +3 -2 Bungled *cut/paste* leaves kernel with *page faults*.. (read all about it!)
101	src/sys/kern/kern_shutdown.c	date: 1996/08/26 21:47:56; author: julian; state: Exp; lines: +3 -28 Remove the old cleanup code as it is no longer used.. also fix two cases of = instead of == (*cut+paste bug duplication*)
102	src/sys/dev/si/si.c	date: 1996/06/16 13:32:16; author: peter; state: Exp; lines: +2 -2 Fix *cut/paste error*; a read-only variable should have been read/write.
103	src/lib/libc/locale/setlocale.c	date: 1995/08/05 17:32:06; author: ache; state: Exp; lines: +3 -3 Fix *cut&paste error*: LC_COLLATE should be LC_TIME

104	src/sys/nfs/nfs_node.c	date: 1995/07/22 03:32:18; author: davidg; state: Exp; lines: +6 -14 Correct my *cut-n-paste* job from ffs_vfsops.c and fix up the formatting to be similar.
105	src/sys/amd64/amd64/trap.c	date: 1995/07/16 14:10:55; author: peter; state: Exp; lines: +2 -6 This fixes a compiler warning, and a cosmetic problem with the linux emul code when compiling with "options KTRACE". ktrsyscall() was expecting an array of integers, this was passing the address of a structure containing an array of integers.. The cosmetic problem was that it was calling the "enter syscall" trace hook twice - this looks like a *cut/paste error/typo*.
106	src/sys/amd64/amd64/trap.c	date: 1995/07/16 14:10:55; author: peter; state: Exp; lines: +2 -6 This fixes a compiler warning, and a cosmetic problem with the linux emul code when compiling with "options KTRACE". ktrsyscall() was expecting an array of integers, this was passing the address of a structure containing an array of integers.. The cosmetic problem was that it was calling the "enter syscall" trace hook twice - this looks like a *cut/paste error/typo*.
107	src/sys/i386/i386/trap.c	date: 1995/07/16 14:10:55; author: peter; state: Exp; lines: +2 -6 This fixes a compiler warning, and a cosmetic problem with the linux emul code when compiling with "options KTRACE". ktrsyscall() was expecting an array of integers, this was passing the address of a structure containing an array of integers.. The cosmetic problem was that it was calling the "enter syscall" trace hook twice - this looks like a *cut/paste error/typo*.

108	src/sys/i386/i386/trap.c	date: 1995/07/16 14:10:55; author: peter; state: Exp; lines: +2 -6 This fixes a compiler warning, and a cosmetic problem with the linux emul code when compiling with "options KTRACE". ktrsyscall() was expecting an array of integers, this was passing the address of a structure containing an array of integers.. The cosmetic problem was that it was calling the "enter syscall" trace hook twice - this looks like a *cut/paste error/typo*.
109	src/usr.sbin/sade/menus.c	date: 2001/03/08 10:41:40; author: jkh; state: Exp; lines: +1 -2 Fix a *paste*-o which introduced a *syntax error*.
110	src/sbin/mdconfig/mdconfig.c	date: 2007/02/27 18:59:27; author: n_hibma; state: Exp; lines: +0 -0 Forced commit (*cut&paste error* in the MFC): The default is '-t swap' not '-t malloc'.
111	src/sys/netinet/ipfw/ip_dummynet.c	date: 2009/12/16 10:48:40; author: luigi; state: Exp; lines: +1 -0 SVN rev 200601 on 2009-12-16 10:48:40Z by luigi - remove some misspelling of names (#define V_foo VNET(bar)) that slipped in due to *cut&paste*
112	src/sys/dev/sk/if_sk.c	date: 2000/11/02 00:04:27; author: wpaul; state: Exp; lines: +3 -3 MFC: fix resource deallocation bugs in these drivers, which have apparently been here a while (and which I *cut&pasted* several times over :/).

113	src/sys/vm/phys_pager.c	date: 2007/11/10 11:23:01; author: remko; state: Exp; lines: +2 -2 branches: 1.23.2.1.4; MFC rev 1.29 phys_pager.c Correct a *copy and paste* in phys_pager.c, we are talking about phys here and not about devices.

Appendix C

SPA

C.1 Example of Type-A inconsistency detection

The example of Table C.1 shows a Type-A inconsistency that SPA detected in the `Eclipse CDT` codebase. Table C.2 summarizes output from each step of the algorithm. Statements `R11` and `R12` from the reference are ported to `T11` and `T12` in the target respectively. `R11` is backward control dependent on `R10`. Similarly, `T11` is backward control dependent on `T10`. Since AST type of `R10` and `T10` are identical and their AST labels are also syntactically same, `T10` and `R10` are isomorphic nodes. `R10` and `T10` are control dependent on `R8` and `T8` respectively, along the `true` control edge. The AST label of `R8` and `T8` are not syntactically identical. Thus, `R8` and `T8` are non-isomorphic. We add `R8` and `T8` to IC_{ref} and IC_{tar} respectively.

Table C.1: Example code from Eclipse CDT showing Type-A inconsistency

Reference Edit : Eclipse CDT File: `ElfHelper.java` , version: CDT_8_1_0

```
 R1. public Elf.Symbol[] getExternalObjects() throws IOException {
 R2.     Vector<Symbol> v = new Vector<Symbol>();
 R3.
 R4.     loadSymbols();
 R5.     loadSections();
 R6.
 R7.     for (int i = 0; i < dynsyms.length; i++) {
 R8.         if (dynsyms[i].st_bind() == Elf.Symbol.STB_GLOBAL &&
      dynsyms[i].st_type() == Elf.Symbol.STT_OBJECT) {
 R9.             int idx = dynsyms[i].st_shndx;
R10.             if (idx < Elf.Symbol.SHN_HIPROC && idx > Elf.Symbol.
    SHN_LOPROC) {
R11.+                 String name = dynsyms[i].toString();
R12.+                if (name != null && name.trim().length() > 0)
R13.                     v.add(dynsyms[i]);
R14.             } else if (idx >= 0 && sections[idx].sh_type == Elf.
    Section.SHT_NULL) {
R15.                 v.add(dynsyms[i]);
R16.             }
R17.         }
R18.     }
R19.
R20.     Elf.Symbol[] ret = v.toArray(new Elf.Symbol[v.size()]);
R21.     return ret;
R22. }
```

Target Edit : Eclipse CDT File: `ElfHelper.java`, version: CDT_8_1_0

```
 T1.public Elf.Symbol[] getLocalObjects() throws IOException {
 T2.     Vector<Symbol> v = new Vector<Symbol>();
 T3.
 T4.     loadSymbols();
 T5.     loadSections();
 T6.
 T7.     for (int i = 0; i < symbols.length; i++) {
 T8.         if ( symbols[i].st_type() == Elf.Symbol.STT_OBJECT) {
 T9.             int idx = symbols[i].st_shndx;
T10.             if (idx < Elf.Symbol.SHN_HIPROC && idx > Elf.Symbol.
    SHN_LOPROC) {
T11.+                String name = symbols[i].toString();
T12.+                if (name != null && name.trim().length() > 0)
T13.                     v.add(symbols[i]);
T14.             } else if (idx >= 0 && sections[idx].sh_type != Elf.
    Section.SHT_NULL) {
T15.                 v.add(symbols[i]);
T16.             }
T17.         }
T18.     }
T19.
T20.     Elf.Symbol[] ret = v.toArray(new Elf.Symbol[v.size()]);
T21.     return ret;
T22.}
```

Ported code begins with +. The red line shows control flow inconsistency found by SPA w.r.t. the ported code.

Table C.2: Type-A inconsistency detection for Table C.1

Steps	**Reference**	**Target**
Edited Nodes	E_r = {R11,R12,R13}	E_t = {T11,T12,T13}
Ported Edits (PM)	{(R11,T11),(R12,T12)}	
Isomorphic Nodes	{(R1,T1),(R2,T2),(R9,T9),(R10,T10), (R11,T11),(R12,T12),(R13,T13),(R14,T14), (R15,T15),(R20,T20)}	
Inconsistency Detection		
Non-Isomorphic control edges	(R8,R10,true) (R8,R9,true)	(T8,T10,true) (T8,T9,true)
Inconsistent Nodes	IC_{ref} = {R8}	IC_{tar} = {T8}

C.2 Example of Type-B1 inconsistency detection

The example of Table C.3 shows an instance of variable renaming inconsistency (Type-B1). A set of reference edits from line `R4` to `R18` are ported to target context (from line `T4` to `T18`). Table C.4 summarizes the identified ported node pairs and the corresponding isomorphic sub-graphs. For each isomorphic node pair, we establish identifier mapping. For example, corresponding to `(R4,T4)`, we map variable `bp` to `rabp`. In fact `bp` is mapped to `rabp` 7 times in the above example ((`R2`,`T2`),(`R4`,`T4`),(`R7`,`T7`),(`R8`,`T8`),(`R9`,`T9`),(`R11`,`T11`),(`R20`,`T20`)). However, node pairs `(R13,T13)` and `(R15,T15)` map the reference variable `bp` to the target variable `bp` 2 times. This raises conflict in the variable mapping. While in 77.77% cases `bp` is mapped to `rabp`, for only 22.22% cases `bp` is mapped to `bp`. In this phase we output such inconsistency.

Table C.3: Adopted code from FreeBSD showing Type-B1 inconsistency

Reference Edit : Adopted from FreeBSD: `vfs_bio.c` , version: 1.351

```
 R1.breadn(...) {
 R2.     bp = getblk(vp, blkno, size, 0, 0);
 R3.
 R4.+     if ((bp.b_flags & B_CACHE) == 0) {
 R5.+         if (curthread != PCPU_GET(idlethread))
 R6.+             curthread.td_prock++;
 R7.+         bp.b_iocmd = BIO_READ;
 R8.+         bp.b_flags &= ~B_INVAL;
 R9.+         bp.b_ioflags &= ~BIO_ERROR;
R10.
R11.+         vfs_busy_pages(bp, 0);
R12.+         if (vp.v_type == VCHR)
R13.+             VOP_SPECSTRATEGY(vp, bp);
R14.+         else
R15.+             VOP_STRATEGY(vp, bp);
R16.+         ++readwait;
R17.+     }
R18.
R19.     if (readwait != 0) {
R20.         rv = bufwait(bp);
R21.     }
R22.     return (rv);
R23.}
```

Target Edit : Adopted from FreeBSD: `vfs_bio.c` , version: 1.351

```
 T1.int breadn(...) {
 T2.     rabp = getblk(vp, rablkno, rabsize, 0, 0);
 T3.
 T4.+     if ((rabp.b_flags & B_CACHE) == 0) {
 T5.+         if (curthread != PCPU_GET(idlethread))
 T6.+             curthread.td_proc.p_stats++;
 T7.+         rabp.b_iocmd = BIO_READ;
 T8.+         rabp.b_flags &= ~B_INVAL;
 T9.+         rabp.b_ioflags &= ~BIO_ERROR;
T10.
T11.+         vfs_busy_pages(rabp, 0);
T12.+         if (vp.v_type == VCHR)
T13.+             VOP_SPECSTRATEGY(vp, bp);
T14.+         else
T15.+             VOP_STRATEGY(vp, bp);
T16.+     }
T17.
T18.
T19.     if (readwait != 0) {
T20.         rv = bufwait(rabp);
T21.     }
T22.     return (rv);
T23.}
```

Table C.4: Type-B1 inconsistency detection for the above example

Steps	**Reference**	**Target**
Edited Nodes	E_r = {R4 - R17}	E_t = {T4 - T18}
Ported Nodes (PM)	{(R4,T4),(R5,T5),(R6,T6),(R8,T8), (R9,T9),(R11,T11),(R12,T12),(R13,T13), (R14,T14),(R15,T15)}	
Isomorphic Nodes	{(R1,T1),(R2,T2),(R4,T4),(R5,T5), ((R6,T6),(R8,T8),(R9,T9),(R11,T11), (R12,T12),(R13,T13),(R14,T14),(R15,T15), (R19,T19),(R20,T20)}	
Inconsistency Detection		
Type-B1	`bp` $\rightarrow$ `bp` : 22.22% `bp` $\rightarrow$ `rabp` : 77.77%	

C.3 Example of Type-C inconsistency detection

The example in Table C.5 shows a Type-C error SPA detected in the `FreeBSD` code base. Table C.6 summarizes output from each step of the algorithm. Statements `R11` and `R18` from the reference are ported to `T11` and `T18` in the target implementation. We add the ported node pairs to PM. The reference node `R12` are backward data dependent on `R1`. The corresponding ported node `T12` is backward data dependent on `T0`. Since AST type of `R1` is a method declaration and AST type of `T0` is variable declaration, the AST types do not match. Thus, `R1` and `T0` are non-isomorphic nodes. We add them to inconsistent set IC_{ref} and IC_{tar} respectively.

Table C.5: An adopted code from FreeBSD showing Type-C inconsistency

Reference Edit : Adopted from FreeBSD: `sched_4bsd.c` , version: 1.90

```
R1.int parse_uuid(String optarg, int uuid)
R2.     {
R3.     int status = 0;
R4.
R5.     uuid_from_string(optarg, uuid, status);
R6.
R7.     if (status == uuid_s_ok)
R8.             return (0);
R9.
R10.
R11.+   if(optarg.startsWith("e")) {
R12.+           if (optarg.compareTo("efi") == 0) {
R13.+                   int efi = GPT_ENT_TYPE_EFI;
R14.+                   uuid = efi;
R15.+           }
R16.+   }
R17.+   else
R18.+           return (EINVAL);
R19.
R20.    return uuid;
R21.}
```

Target Edit : Adopted from FreeBSD: `sched_4bsd.c` , version: 1.90 Ported

```
T0.String optarg;
T1.int parse_uuid(String s, int uuid)
T2.     {
T3.     int status = 0;
T4.
T5.     uuid_from_string(s, uuid, status);
T6.
T7.     if (status == uuid_s_ok)
T8.             return (0);
T9.
T10.
T11.+   if(s.startsWith("e")) {
T12.+           if (optarg.compareTo("efi") == 0) {
T13.+                   int efi = GPT_ENT_TYPE_EFI;
T14.+                   uuid = efi;
T15.+           }
T16.+   }
T17.+   else
T18.+           return (EINVAL);
T19.
T20.    return uuid;
T21. }
```

code fragments begin with +. The red line shows data flow inconsistency found by SPA w.r.t. the ported code.

Table C.6: Type-C inconsistency detection for the example in Table C.5

Steps	**Reference**	**Target**
Edited Nodes	E_r = {R11 - R18}	E_t = {T11 - T18}
Ported Edits (PM)	{(R11,T11),(R12,T12),(R13,T13),(R14,T14), (R15,T15),(R16,T16),(R17,T17),(R18,T18)}	
Isomorphic Nodes	{(R1,T1),(R3,T3),(R5,T5),(R7,T7),(R11,T11),(R12,T12), (R13,T13),(R14,T14),(R17,T17),(R18,T18),(R20,T20)}	
Inconsistency Detection		
Non-isomorphic data dependences	(R1,R12,optarg)	(T0,T12,optarg)
Inconsistent Nodes	IC_{ref} = {R1}	IC_{tar} = {T0}

C.4 Example of Type-D inconsistency detection

The example in Table C.7 shows a Type-D inconsistency SPA detected in the `FreeBSD` codebase. Table C.8 summarizes the output of the algorithm. Statements `R5` to `R13` from the reference are ported to lines `T15` to `T23` in the target respectively. Here, variable `rx_big` should be renamed to `rx_small`. However, not a single instance of `rx_bug` is renamed; hence we cannot detect this error using our Type-B1 inconsistency detection algorithm. In fact, this inconsistency can be detected using Type-D diagnosis algorithm because the incorrect porting generates a redundancy. The developers log also suggested this—*Fix a mbuf leak caused by a cut&paste bug where the small ring's mbufs were never freed, but the big ring was freed twice*.

Since ported node `T15` is backward data dependent on `T1`, `T1` is in the

Table C.7: An adopted code from FreeBSD showing Type-D inconsistency

Reference Edit : Adopted from FreeBSD: `if_mxge.c` , version: 1.27

```
 R1. void mxge_free_mbufs(mxge_softc_t sc)
 R2. {
 R3.  int i;
 R4.
 R5.+  for (i = 0; i <= sc.rx_big.mask; i++) {
 R6.+      if (sc.rx_big.info[i].m == null)
 R7.+          continue;
 R8.+
 R9.+      bus_dmamap_unload(sc.rx_big.dmat,
R10.+              sc.rx_big.info[i].map);
R11.+      m_freem(sc.rx_big.info[i].m);
R12.+      sc.rx_big.info[i].m = null;
R13.+  }
R14.
R15.  }
```

Target Edit : Adopted from FreeBSD: `if_mxge.c` , version: 1.27

```
 T1. void mxge_free_mbufs(mxge_softc_t sc)
 T2. {
 T3.  int i;
 T4.
 T5.  for (i = 0; i <= sc.rx_big.mask; i++) {
 T6.      if (sc.rx_big.info[i].m == null)
 T7.          continue;
 T8.
 T9.      bus_dmamap_unload(sc.rx_big.dmat,
T10.              sc.rx_big.info[i].map);
T11.      m_freem(sc.rx_big.info[i].m);
T12.      sc.rx_big.info[i].m = null;
T13.  }
T14.
T15.+  for (i = 0; i <= sc.~~rx_big~~rx_small.mask; i++) {
T16.+      if (sc.~~rx_big~~rx_small.info[i].m == null)
T17.+          continue;
T18.+
T19.+      bus_dmamap_unload(sc.~~rx_big~~rx_small.dmat,
T20.+              sc.~~rx_big~~rx_small.info[i].map);
T21.+      m_freem(sc.~~rx_big~~rx_small.info[i].m);
T22.+      sc.~~rx_big~~rx_small.info[i].m = null;
T23.+  }
T24.
T25.  }
```

The ported edits begin with +. The red line shows data flow inconsistency found by SPA w.r.t. the ported edits.

Table C.8: Type-D inconsistency detection in Table C.7

Edited Nodes	E_{ref} = {R5 - R13}	E_{tar} = {T15 - T23}
Ported Nodes (PM)	{(R5,T15),(R6,T16),(R9,T19), (R10,T20),(R11,T21),(R12,T22)}	
Isomorphic Nodes	{(R1,T1),(R5,T15),(R6,T16),(R9,T19)	
C_{tar}	{T1,T15,T16,T19,T20,T21,T22}	
Inconsistency Detection		
$\Delta\_redundancy$	{ (T1,T5,NA),(T1,T15,NA)}	
Inconsistent Nodes	IC_{ref} = -	IC_{tar} = {T5, T15}

context of the ported edit. `T1` controls `T5` and `T15`, which have `identical` AST labels and types. As identical nodes `T5` and `T15` are under the same control predicate `T1`, SPA diagnoses a Type-D inconsistency.

Bibliography

[1] C++ to java converter: http://www.tangiblesoftwaresolutions.com.

[2] Crystal a static analysis framework for education and research http://code.google.com/p/crystalsaf/.

[3] Freebsd history: http://en.wikipedia.org/wiki/freebsd.

[4] http://git-scm.com/.

[5] http://mercurial.selenic.com/.

[6] Netbsd history: http://en.wikipedia.org/wiki/netbsd.

[7] Openbsd history: http://en.wikipedia.org/wiki/openbsd.

[8] George W. Adamson and Jillian 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.

[9] E. Adar and Miryung Kim. Softguess: Visualization and exploration of code clones in context. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 762 –766, may 2007.

[10] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *Empirical*

Software Engineering, 2005. 2005 International Symposium on, page 10 pp., nov. 2005.

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

[12] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[14] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Brenda S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, March 1992.

[16] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Pro-*

ceedings of the International Conference on Software Maintenance, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[17] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. Social interactions around cross-system bug fixings: the case of freebsd and openbsd. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 143–152, New York, NY, USA, 2011. ACM.

[18] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[19] James 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.

[20] James R Cordy. Exploring large-scale system similarity. using incremental clone detection and live scatterplots. In *ICPC 2011, 19th International Conference on Program Comprehension (to appear)*, 2011.

[21] Massimiliano Di Penta, Daniel M. German, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 145–154, New York, NY, USA, 2010. ACM.

[22] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall. Mining evolution data of a product family. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[24] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[25] Zachary P. Fry and Westley Weimer. A human study of fault localization accuracy. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[26] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[27] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[28] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 175–190, New York, NY, USA, 2010. ACM.

[29] Daniel M. German, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 81–90, Washington, DC, USA, 2009. IEEE Computer Society.

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

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

[32] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.

[33] Patricia Jablonski and Daqing Hou. Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 16–20, New York, NY, USA, 2007. ACM.

[34] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Icse '07: Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, New York, NY, USA, 2007. ACM.

[36] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[37] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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.

[38] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[39] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE*

'04: Proceedings of the 2004 International Symposium on Empirical Software Engineering, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Miryung Kim, Vibha Sazawal, David Notkin, and Gail 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, New York, NY, USA, 2005. ACM.

[41] Miryung Kim, Vibha Sazawal, David Notkin, and Gail 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.

[42] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–169, New York, NY, USA, 2000. ACM Press.

[43] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, UK, 2001. Springer-Verlag.

[44] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[45] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–, Washington, DC, USA, 2001. IEEE Computer Society.

[46] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[47] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: a language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[48] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[49] Jingyue Li and Michael D. Ernst. Cbcd: cloned buggy code detector. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 310–320, Piscataway, NJ, USA, 2012. IEEE Press.

[50] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[51] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society.

[52] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 329–342, New York, NY, USA, 2011. ACM.

[53] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.

[54] Audris Mockus and Lawrence 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.

[55] Nachiappan Nagappan and Thomas 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.

[56] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 70–79, New York, NY, USA, 2012. ACM.

[57] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien 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, New York, NY, USA, 2010. ACM.

[58] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Clone-aware configuration management. In *Automated Software Engineeering*, 2009.

[59] Jeroen Ooms. Possible directions for improving dependency versioning in r. 2013.

[60] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 128–137, New York, NY, USA, 2003. ACM.

[61] Jihun Park, Miryung Kim, B. Ray, and Doo-Hwan Bae. An empirical study of supplementary bug fixes. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 40 –49, june 2012.

[62] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, New York, NY, USA, 2008. ACM. differential symbolic execution.

[63] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.

[64] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.

[65] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[66] Baishakhi Ray and Miryung Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 53:1–53:11, New York, NY, USA, 2012. ACM.

[67] Baishakhi Ray, Christopher Wiley, and Miryung Kim. Repertoire: A cross-system porting analysis tool for forked software projects. In *FSE-20: ACM SIGSOFT the 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, to appear.

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

[69] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004. ACM.

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

[71] E. M. Riseman and A. R. Hanson. A contextual postprocessing system for error correction using binary n-grams. *IEEE Trans. Comput.*, 23(5):480–493, May 1974.

[72] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 499–510, 2007.

[73] Neha Rungta, Suzette Person, and Joshua Branchaud. A change impact analysis to characterize evolving program behaviors. In *ICSM*, pages 109–118, 2012.

[74] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

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

[76] Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, eclipse '06, pages 50–54, New York, NY, USA, 2006. ACM.

[77] Xiaoyin Wang, Yingnong Dang, Lu Zhang, Dongmei Zhang, Erica Lan, and Hong Mei. Can i clone this piece of code here? In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 170–179, New York, NY, USA, 2012. ACM.

[78] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 461–470, New York, NY, USA, 2008. ACM.

[79] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[80] Wikipedia. List of software forks — Wikipedia, the free encyclopedia, 2012.

[81] Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro 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.

[82] Xin Yan and Xiao Gang Su. *Linear Regression Analysis: Theory and Computing*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2009.

[83] Wuu Yang. Identifying syntactic differences between two programs. *Software – Practice & Experience*, 21(7):739–755, 1991.

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

Vita

Baishakhi Ray is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Texas at Austin. Her research focuses on software engineering, in particular, automated program analysis algorithms and tools for evolving software. Baishakhi was born in Kolkata, India. She earned her bachelor's degree in 2004 from Calcutta University, with majors in Physics and Computer Science. She completed her masters in Computer Science from University of Colorado, Boulder in 2009. She worked in various software companies including Texas Instruments, Ericsson, Avaya, etc. In summer 2012, she also took part in Google Summer Code.

Permanent address: AJ 100, Sector-II, Saltlake City.
Kolkata, India 700091

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.