Detecting and Characterizing Semantic Inconsistencies in Ported Code

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

Suzette Person
NASA Langley Research Center
Hampton, USA
suzette.person@nasa.gov

Neha Rungta
NASA Ames Research Center
Mountain View, USA
neha.s.rungta@nasa.gov

Abstract—Adding similar features and bug fixes often requires porting program patches from reference implementations and adapting them to target implementations. Porting errors may result from faulty adaptations or inconsistent updates. This paper investigates (1) the types of porting errors found in practice, and (2) how to detect and characterize potential porting errors. Analyzing version histories, we define five categories of porting errors, including incorrect control- and data-flow, code redundancy, inconsistent identifier renamings, etc. Leveraging this categorization, we design a static control- and data-dependence analysis technique, SPA, to detect and characterize porting inconsistencies. Our evaluation on 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% to 100% recall. In a comparison with two existing error detection tools, SPA improves precision by 14 to 17 percentage points.

I. INTRODUCTION

Developers often port code from one implementation to another in order to implement similar features or bug fixes. A recent case study of OpenBSD, NetBSD, and FreeBSD found that 11% to 16% code changes are ported from peer projects [18]. Also, when libraries and frameworks evolve their APIs, client applications make similar updates to use the new APIs correctly [3]. In a large code base, typically 10% to 30% of the code is considered as code clones [11], which often require similar updates during software evolution [13]. When porting changes from one implementation to another, developers generally need to adapt the ported changes to fit the new context. The code in the reference often serves as a template that is pasted into the target implementation, and then later adapted [12].

The process of adapting a change to fit another context can be error-prone, often resulting in porting errors. Chou et al. report that a significant portion of system bugs comes from ported edits [4]. In a case study of clone related bugs, Juergens et al. discover that “nearly every second, unintentional changes to clones lead to a fault” [10]. Li et al. identify errors in Linux and FreeBSD resulting from developers forgetting to rename identifiers after porting code [15]. Jiang et al. [9] present evidence of porting errors when similar code appears in different contexts. Porting errors can also happen when developers evolve ported code differently [6], [10].

When developers port code from a reference to a target context, they usually expect the ported code to behave similarly. Existing tool support for detecting semantic inconsistencies in ported code is limited. For example, Li et al. and Juergens et al. find inconsistent clones using a lexical clone detection analysis [10], [15]. Jiang et al. and Gabel et al. report clone related bugs by comparing the syntax tree structures for two clones [6], [9]. Such syntactic and lexical analyses are not sufficient to detect the semantic inconsistencies arising from updates to the ported code in different contexts.

The goal of this work is to assist developers in porting edits from one context to another, by detecting semantic inconsistencies that may indicate a porting error. As a first step towards this goal, we study the extent and characteristics of porting errors that occur in practice to better understand the types of porting errors and their fixes. In our study, we work backwards by first mining the version histories of Linux and FreeBSD to detect commit messages containing porting error related keywords. We then analyze three types of source code commits—fix-inducing, error-inducing, and reference—and their corresponding patches. A patch is the set of program statements that are added, deleted, or modified in a program version with respect to its previous version. Note that modified statements can also be represented as deleted statements in the old version and added statements in the new version. We use Silverski et al.’s fix-inducing change identification method [21] to identify the patch that originally introduced the porting error. We then use Repertoire [18] to find a reference patch that served as the template for the error-inducing patch. Through manual investigation of the reference patch, the error-inducing patch, and the fix patch, we find that many of the porting errors result from incorrect adaptation of the ported code, including inconsistent identifier renamings, different control- and data-flow contexts in the reference and target implementations, and code redundancy.

Leveraging this 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 program statements surrounding the ported code. SPA takes two code patches as input: a reference patch (Ref_{old} and Ref_{new}) and a target patch (Tar_{old} and Tar_{new}). 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. 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, and compare the results with two state-of-the-art tools, DejaVu [6] and Jiang et al.’s clone related bug detection tool [9]. 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% to 100% recall. SPA outperforms two related error detection tools with a precision improvement of 14 to 17 percentage points.

We make the following contributions:

- We conduct a comprehensive study of the extent and characteristics of porting errors reported for real-world systems. We identify categories of common porting errors related to inconsistent control flow, inconsistent data flow, inconsistent identifier renaming, and code redundancy.
- Leveraging information about commonly found porting errors, we design and implement a novel algorithm, SPA, to detect potential porting errors based on inconsistent semantics of ported code between the reference and target contexts.
- We conduct an empirical evaluation of SPA’s ability to detect and characterize porting inconsistencies in four large open-source codebases.

The rest of the paper is organized as follows. Section II discusses an empirical study of porting errors in Linux and FreeBSD. Section III discusses SPA’s methodology for detecting and characterizing porting inconsistencies. Section IV presents an empirical evaluation of SPA’s capability to detect and characterize porting inconsistencies. Section V discusses related work. Finally, Section VI summarizes our work and directions for future work.

II. AN EMPIRICAL STUDY OF 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. 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. In the remainder of this section, we present the study setup, results, and a description of the five categories of porting errors. We first define several key terms used in this work.

**Definition 2.1:** 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 2.2:** 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 are also edited similarly.

**Definition 2.3:** Context of ported code is the set of program statements in a method that are not part of the ported code.

A. Study Method

We mine the commit logs and analyze version histories for Linux and FreeBSD. Table I shows the size of the two projects in KLOC, the evolution period under study, and the number of unique developers who made commits during that period.

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. A sample commit message in FreeBSD is "Fix cut/paste bug which would result in a panic " The corresponding code patch fixes the porting error.

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 IR-1 example in Table II. 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 Siwerski et al. [21] identify a fix-inducing patch. We then use the REPETTORE tool to identify a set of candidate reference patches that may serve as the template for the target patch \( p_t \) [19]. 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 IR-1 example in Table II where the developer forgot to update an identifier \( \text{bp} \) to \( \text{r@bp} \) after porting code fragments from the reference patch, we expect the reference patch to contain the unaltered code fragment related to \( \text{bp} \). When multiple patches contain similar unaltered code fragments, we select a patch with the maximum number of similar lines.

B. Porting Errors Characterization

In our study we were able to identify 113 and 182 porting errors documented in FreeBSD and Linux version histories over the course of 18 years and 3 years respectively. Based on the porting errors analyzed in our study, we were able to classify the errors into five different categories. We use the
code snippets in Table II to discuss each of the categories of porting errors below.

### TABLE I

<table>
<thead>
<tr>
<th>KLOC</th>
<th>developers</th>
<th>years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>14,998</td>
<td>6,839</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>4,479</td>
<td>405</td>
</tr>
</tbody>
</table>

**ICF: Inconsistent Control Flow.** 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 ICF example shown in Table II, there is an extra for loop, highlighted in gray, in the reference context. Thus, the continue statement in the reference code 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.

**IR: Inconsistent Renaming.** 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 subcategories:

**IR-1: Inconsistent renamings of identifiers.** 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 \( rbp \) three times, missing the instances marked in red in the IR-1 example in Table II.

**IR-2: Inconsistent renamings of related identifiers.** Developers consistently rename an identifier, but forget to update all related identifiers. In the IR-2 example in Table II, all instances of the OFDM related macro \( IWL\_FIRST\_OFDM\_RATE \) are updated to the CCK related macro \( IWL\_FIRST\_CCK\_RATE \). However, 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.

**IDF: Inconsistent Data Flow.** This inconsistency occurs when developers mistakenly insert code to a different data initialization context. In the IDF example in Table II, the first argument of the \( \\text{strcmp} \) method \( \text{optarg} \) is initialized differently in the reference and target edits. \( \text{optarg} \) is an environment variable initialized by the \( \text{getopt} \) call that parses the command line arguments and stores the next argument to \( \text{optarg} \). Hence, the function call \( \text{getopt} \); and the use of variable \( \text{optarg} \) should occur as a pair. In the reference context, \( \text{optarg} \) is used after \( \text{getopt} \); and thus is initialized properly. In the target context, however, there is no call to \( \text{getopt} \). Thus, \( \text{optarg} \) is not initialized properly.

**RDN: 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 RDN example in Table II, a code fragment related to \( \text{memcpy} \) was ported to the same function body twice under the same scope in FreeBSD. The corresponding patch removes \( \text{memcpy} \) and the buffer initialization statements to correct the redundant operations.

**OTH: 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/striped, version 1.3, a comment related to “Concat Name” was updated not to “Strip Name”.

### C. Distribution of Porting Errors in FreeBSD and Linux

<table>
<thead>
<tr>
<th></th>
<th>ICF</th>
<th>IR</th>
<th>IDF</th>
<th>RDN</th>
<th>OTH</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>23%</td>
<td>74%</td>
<td>26%</td>
<td>47%</td>
<td>25%</td>
<td>182%</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>9%</td>
<td>54%</td>
<td>32%</td>
<td>14%</td>
<td>28%</td>
<td>113%</td>
</tr>
</tbody>
</table>

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 III 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 (IR)—47.78% and 40.66% in FreeBSD and Linux respectively. The errors related to control (ICF) and data (IDF) flow inconsistency make up more than 25% of the total porting errors. The rest of the errors are either due to redundant operations (RDN)—12.39% and 25.82%, or wrong indentation and comments (OTH)—24.78% and 13.74% in FreeBSD and Linux respectively.

The error categories are not mutually exclusive. For example, an inconsistent renaming error (IR) may also cause an inconsistent data initialization error (IDF)—17.7% and 1.6% of the porting errors in FreeBSD and Linux respectively are both types IR and IDF. An inconsistent data initialization error (IDF) may also generate redundant operations (RDN)—1.8% in FreeBSD and 2.7% in Linux. Sometimes, an inconsistent control flow (ICF) may also initialize the data erroneously.
### Table II: Examples of Porting Errors of Different Types

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Source Code Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICF: Inconsistent Control Flow</td>
<td>Fix a copy-paste bug in NON KSE case.</td>
<td>FreeBSD commit: src/sys/kern/sched_4bsd.c, version 1.90, Author: davidz, Date: 2006/11/14</td>
</tr>
<tr>
<td>IR-1: Inconsistent renamings of identifiers</td>
<td>Fix cut-and-paste bug which would result in a panic because buffer was being bidone/multiple times.</td>
<td>FreeBSD commit: src/sys/kern/vfs_bio.c, version 1.351, Author: phil, Date: 2003-01-05</td>
</tr>
<tr>
<td>IR-2: Inconsistent renamings of related identifiers</td>
<td>Fix a copy paste error in iwLcalc_basic_rates which leads to a wrong calculation of CCK basic rates.</td>
<td>Linux commit: 5ed08946a00e6bd0364a365432b0a460b818a2, Author: Emmanuel Grumbach, Date: 2013-11-20</td>
</tr>
<tr>
<td>IDF: Inconsistent Data Flow</td>
<td>Fix cut and paste bug: compare arguments against known aliases, not the global optarg.</td>
<td>FreeBSD commit: src/sbin/gpt/gpt.c, version 1.16, Author: macel, Date: 2006-07-07</td>
</tr>
<tr>
<td>RDN: Redundant operations</td>
<td>Looks like a copy-and-paste error, identical lines are a few lines below the ones removed. ...</td>
<td>Linux commit: f9e2f1b68f18354f82f8c75c326a0f6537c82a7e2, Author: John W. Linville, Date: 2011-04-29</td>
</tr>
</tbody>
</table>

Ported lines start with "*". The errors are marked in red. The fixes are highlighted in green.
We illustrate the SPA approach with an example shown in Table IV. The example is an adapted version of code fragments from FreeBSD. The 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 T5 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.

The program statements that are changed between the old and new versions are highlighted in gray and the ported edits are marked with “+” in Table IV. Ported edits T5, T10 and T13 in `Tar_new` correspond to R4, R5 and R6 in `Ref_new` respectively. The ported edits in `Tar_new` are control-dependent on T4 and data-dependent on T1, T2 and T12. Also T11, T14 and T15 are data-dependent on the ported edits T10 and T13. All of these statements are treated as impacted statements. Similarly, R1, R2 and R8 are marked as impacted statements in `Ref_new`. Next, we present the details of how impacted ported nodes are generated.

B. Identify the Impact of the Ported Code

We present the three main steps to identify the porting context that may impact or be impacted by 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`: `ptar := Δ(Tar_old, Tar_new)` and `pref := Δ(Ref_old, Ref_new)`.

1. Identify Edits in the Reference and Target: 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` [5]. This algorithm is inspired by Meng et al.’s edit script generation and extends its implementation [16, 17]. For the code shown in Table IV, three edit (insert) operations are identified in the reference patch, and five edit operations are identified in the target patch. SPA uses the edit operations to generate 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 an edited statement in a program patch `p`. The source lines corresponding to the edited nodes are highlighted using a gray background in Table IV.

2. Identify Ported Nodes: SPA determines the correspondence of statements in the ported code between 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 ∈ E_ref` and `t ∈ 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

---

1 Type OTH (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.
The output ranges from delete to loop by PNP strings are either identical or very similar, i.e., when labels are also similar above a certain threshold based on bi-gram similarity [20]. 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.

$$\text{PNP} = \{(r,t) | r \in E_{\text{ref}} \land t \in E_{\text{tar}} \land \text{clone}(r,t)\}$$  \hspace{1cm} (1)

PNP is a set of ported node pairs where each pair \((r,t)\) \in PNP represents a node ported from a reference patch to a target patch as defined in Equation 1. Each node in the pair \((r,t)\) is referred to as a {	extit{ported node}}. For example, the nodes corresponding to statements \(R5\) and \(T10\) in Table IV have the same AST node type (declaration) and label (size = count + stats.sizeof ), hence clone\((R5,T10)\) is true and \(R5,T10\) is a ported node pair. However, no AST nodes in \(E_{\text{ref}}\) are syntactically similar to the AST node corresponding to statement \(T11\) in \(E_{\text{tar}}\). Therefore, \(T11\) is not a member of any ported node pairs. All of the statements identified with \(*\) in Table IV have corresponding AST nodes in PNP.

Step 3. Identify Impacted Nodes: Next, SPA identifies the AST nodes in \(Ref_{\text{new}}\) and \(Tar_{\text{new}}\) 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 or may be affected by the ported nodes. SPA identifies the impacted nodes using static intra-procedural data- and control-dependence analyses [22] with respect to the ported nodes. This step bears resemblance to how Sydit identifies the context of edit operations using control and data analysis [16].

Data Dependence. Statement \(S2\) is data dependent on \(S1\), if \(S1\) defines a variable \(v\) and \(S2\) uses \(v\), such that there exists a path from \(S1\) to \(S2\) along which \(v\) is not killed (redefined).

Control Dependence. Statement \(S2\) is control dependent on \(S1\), if execution of \(S2\) depends on the decision made at \(S1\).

Definition 3.1: 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 using 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 \(Ref_{\text{new}}\) and \(Tar_{\text{new}}\) are
derived from their respective program dependence graphs, PDC_{ref} and PDC_{tar}. Given a set of vertices mapping to the ported nodes V_p \subseteq \text{Ref}_{\text{new}} and the PDG for \text{Ref}_{\text{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 find I_{tar} from V_p \subseteq \text{Tar}_{\text{new}}. The vertices corresponding to statements T6 and T7 in Table IV are not control or data dependent on ported code, hence they are not in the impact set.

C. Detect and Categorize Porting Inconsistencies

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

ICF: Inconsistent Control Flow. To detect ICF inconsistencies, 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 \(v_r \in \text{CDG}_{\text{ref}}\) and \(v_t \in \text{CDG}_{\text{tar}}\), is isomorphic if (i) the vertex labels have identical AST types and similar syntactic structures (e.g., nodes 'a = a + b' and 'x = y + z' have same AST type and syntactic structure), 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 [14] to construct the isomorphic sub-graphs.

- 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 it is reachable from \(r \) (\(t \)) in CDG_{ref} (CDG_{tar}), but it is not contained in the respective isomorphic subgraph.

The nodes corresponding to statements R4 and T9 in Table IV 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 although it does not have a corresponding node in the reference.

IR: Inconsistent Renaming. 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 an IR 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.

Table V shows an example of IdMap_{ref} generated from Table IV. SPA generates a map entry \(\text{osflstatfs} \rightarrow \text{ostatfs} \) from the method signatures and \(\text{osflstatfs} \rightarrow \text{ostatfs} \) from the isomorphic nodes R4 and T9. Since the reference variable osflstatfs maps to two target variables, osflstatfs and ostatfs, an IR inconsistency is detected.

### Table V

<table>
<thead>
<tr>
<th>Isomorphic Nodes</th>
<th>Identifier Map (IdMap_{ref})</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{R4}, \text{T9}))</td>
<td>flags \rightarrow flags (1), bufsize \rightarrow bufsize (1)</td>
</tr>
<tr>
<td>((\text{R5}, \text{T10}))</td>
<td>count \rightarrow count (1), bufsize \rightarrow bufsize (2)</td>
</tr>
</tbody>
</table>

The inconsistent mapping is highlighted in red.

Sometimes developers forget to update related identifiers, as shown in the IR-2 example in Table II. 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.

IDF: Inconsistent Data Flow. IDF inconsistency detection is similar to our ICF diagnosis but uses data dependence graphs (DDG) instead of CDGs.

In Table IV, R6 and T13 are statements corresponding to 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, the node corresponding to T12 is added to IC_{tar}.

Similarly, R5 and T11 are statements corresponding to a ported node pair, and both define variable size. However, in the reference implementation, size is used at statement R6, while in the target implementation, size is used at statements T11, T13, and T14. Although R6 and T13 are isomorphic, T11 and T14 create additional data dependences in the target implementation that are not present in the reference implementation. Therefore, the nodes corresponding to T11 and T14 are added to IC_{tar}.

RDN: Redundant operations. To detect redundant ported code, 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 an RDN inconsistency in Tar_{new}. In Table VI, 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 nodes corresponding to statements T13 and T14 as redundant.
Table VI shows the nodes that are inconsistent with respect to the ported code in Table IV, along with their corresponding inconsistency types.

| Inconsistent Control Dependent Nodes (ICD) | T4 |
| Inconsistent Identifier Renaming (IR)      | T9 (identifier: ostatsf) |
| Inconsistent Data Dependent Nodes (IDD)    | T11, T12, T14 |
| Redundant Nodes (RDN)                      | T13, T14 |

### D. Implementation

SPA is implemented using several existing tool chains. First, we extend LASER [17] and Sydit [16], which extract edit scripts to automate systematic program changes. 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. The dependency analysis uses *crystal* [2], a static analysis framework to analyze Java source code.

### IV. 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 [9] and DejaVu [6]. 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 on control and data flow dependences with the ported code.

We also compute SPA’s accuracy to characterize potential inconsistencies based on the categories defined in Section II. 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?

#### A. Study Subjects

To evaluate SPA, we use porting examples from four different projects: FreeBSD, Linux, Eclipse CDT, and Mozilla. Except for Mozilla, the reference and target patches for each artifact are computed using REPertoire [18]. From these, we randomly select (a) 20 examples from FreeBSD, (b) 10 examples from Linux, (c) 60 examples from Eclipse CDT that are ported from CDT versions CDT_2_0_C to CDT_8_1_1, and (d) 42 Mozilla examples from the annotated data set of copy-paste errors provided by Gabel et al. [6]. The FreeBSD and Linux artifacts are from the data sets used in Section II. To retrieve a large number of porting instances, we choose CDT_2_0_C and CDT_8_1_1 versions which are 98 months apart. The Mozilla examples were obtained from DejaVu’s annotated data set, 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 C and C++ porting examples from Linux, FreeBSD and Mozilla examples using a free C/C++ to Java code converter [1].

#### B. Study Methodology

We measure SPA’s capability to detect and categorize porting errors in terms of precision and recall. For each error type $e$ defined in Section II, 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.

#### C. Study Results and Discussions

**RQ1.** Can SPA accurately detect porting inconsistencies?

We compare SPA’s ability to detect porting inconsistencies with Jiang et al.’s clone related bug detection algorithm [9] and DejaVu [6]. Table VII 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 considerably over Jiang et al. SPA improves the precision from 48% to 65%, and marginally 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

---

3[http://www.cse.ucsd.edu/~gabel/research/dejavu_mozilla.zip](http://www.cse.ucsd.edu/~gabel/research/dejavu_mozilla.zip)

3Jiang et al.’s clone detector Deckard and the associated clone bug detector were downloaded from [https://github.com/skyhover/Deckard](https://github.com/skyhover/Deckard).
in detecting errors on the Mozilla data set is 73.53% as shown in Table VII. Because this data set does not contain any examples where DejaVu fails to report an inconsistency, we are unable to assess the number of false negatives for either DejaVu or 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 three tools incorrectly detect inconsistencies include porting code from a while context to a for context, porting code from an if context to a switch-case context, etc.

### TABLE VII

**Inconsistency detection results for Eclipse CDT and Mozilla**

<table>
<thead>
<tr>
<th></th>
<th>Eclipse CDT</th>
<th>Mozilla DejaVu</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPA</td>
<td>Jiang’s Tool</td>
</tr>
<tr>
<td>Detected</td>
<td>43</td>
<td>34</td>
</tr>
<tr>
<td>False Positive</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>False Negative</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>Precision</td>
<td>65.11%</td>
<td>73.53%*</td>
</tr>
<tr>
<td>Recall</td>
<td>90.32%</td>
<td>59.52%*</td>
</tr>
</tbody>
</table>

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

### RQ2. Can SPA accurately categorize different types of porting inconsistencies?

Table VIII shows the precision and recall for SPA in categorizing potential porting errors in FreeBSD and Linux for the error types ICF, IR-1, IR-2, IDF, and RDN. SPA has precision ranging from 50% for ICF to 100% for RDN. The recall for SPA ranges from 62.5% for RDN to 100% for ICF and IDF w.r.t. the porting errors reported in the version histories (see 2nd row in Table VIII). 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 by manual inspection of individual patches. The comparison against this annotated set is shown in Rows 5-7 in Table VIII.

Table IX summarizes 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 ICF, 16% are IR-1, 12% are IR-2, and 40% are IDF. In Mozilla, SPA detects 54 instances of porting inconsistencies, of which 28%, 22%, 7%, and 43% are of type ICF, IR-1, IR-2, and IDF respectively. No RDN 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 ICF 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 IR-1 when the relative ordering of program variables is changed, but the semantics remain unchanged, e.g., a statement $x = y + x$ in the reference implementation is modified to $x = y + x$ in the target. When characterizing IR-2 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 the case of IDF 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, there are some success stories. A bug was fixed in FreeBSD source file: `src/sys/dev/inf/myq/inf_q.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 an RDN bug, which is also confirmed by the developers and took 26 releases and 342 days to detect and fix. Jiang et al.’s tool is not able to detect this bug since it does not handle redundancy.

Another identifier renaming bug was fixed in Linux at commit id 2b9460. Code was ported from method `nlx4_ib_post_send to nlx4_ib_post_recv`, but variable `send_cq` was never updated to `recv_cq`. This bug caused a queue overflow in the `infiband` driver module (a high-speed network driver) and took 974 days to fix. SPA
V. RELATED WORK

Juergens et al. [10] conduct an empirical study on the impact of inconsistent clones in a code base. They detect inconsistent clones using a suffix-tree based, lexical clone detection algorithm. 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.”

Chou et al. show that porting is an important source of bugs in operating systems [4]. 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 [15]. An incorrect adaptation of ported code often leads to porting errors [9]. This observation is aligned with our findings—where we find 113 and 182 porting errors by mining FreeBSD and Linux version histories respectively.

Using CP-Miner, a mining based clone detection tool, Li et al. find 28 and 23 errors in Linux and FreeBSD respectively, which developers created by forgetting to rename identifiers consistently after copy and paste [15]. Jablonski et al. [7] 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 IR inconsistencies, 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 (IR-1 and IR-2).

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.

Jiang et al. show that an inconsistent context can also cause porting errors [9]. 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 [8], and then detect inconsistencies by comparing the contexts. While their diagnosis partially overlaps with our categorization of porting errors (ICF and IR-1), they do not report renaming errors on groups of identifiers (IR-2), data flow inconsistencies (IDF), or redundant operations (RDN).

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. SPA reports 17 percentage point better precision and 3 percentage point 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 [6]. As shown in our evaluation, SPA’s error detection still outperforms DejaVu with 14 percentage point better precision. Also, DejaVu does not report potential error types, while SPA automatically characterizes the detected inconsistencies to help developers detect porting errors.

VI. CONCLUSION

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. By analyzing the version histories for Linux and FreeBSD, we identify five common categories of porting errors, and then use this categorization to design SPA, a novel algorithm to detect and characterize semantic inconsistencies in ported code. 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 the precision of two state-of-the-art techniques with 14 to 17 percentage point.

As part of our future work, we plan to investigate methods for further reducing false positives, such as comparing the dynamic program behaviors of ported code. Based on the observation that not all inconsistencies lead to an error, we also plan to investigate heuristics to rank the inconsistencies based on their error potential. Finally, we plan to integrate SPA with an integrated development environment so that developers can detect porting inconsistencies during the porting process.

ACKNOWLEDGMENT

We 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. This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, SHF-0910818, and CNS-1230498.
REFERENCES


