Recommending Code Changes for Automatic Backporting of Linux Device Drivers

Ferdian Thung, Dinh Xuan Bach Le, David Lo, Julia Lawall

Recommending Code Changes for Automatic Backporting of Linux Device Drivers

Ferdian Thung, Xuan-Bach D. Le, David Lo, and Julia Lawall
School of Information Systems, Singapore Management University, Singapore
Sorbonne Universités, Inria, UPMC, LIP6, France
{ferdiant.2013,dxb.le.2013,davidlo}@smu.edu.sg, Julia.Lawall@lip6.fr

***Abstract*—Device drivers are essential components of any operating system (OS). They specify the communication protocol that allows the OS to interact with a device. However, drivers for new devices are usually created for a specific OS version. These drivers often need to be backported to the older versions to allow use of the new device. Backporting is often done manually, and is tedious and error prone. To alleviate this burden on developers, we propose an automatic recommendation system to guide the selection of backporting changes. Our approach analyzes the version history for cues to recommend candidate changes. We have performed an experiment on 100 Linux driver files and have shown that we can give a recommendation containing the correct backport for 68 of the drivers. For these 68 cases, 73.5%, 85.3%, and 88.2% of the correct recommendations are located in the Top-1, Top-2, and Top-5 positions of the recommendation lists respectively. The successful cases cover various kinds of changes including change of record access, deletion of function argument, change of a function name, change of constant, and change of if condition. Manual investigation of failed cases highlights limitations of our approach, including inability to infer complex changes, and unavailability of relevant cues in version history.**

***Index Terms*—Backporting, Recommendation System, Linux, Device Drivers**

I. Introduction

The Linux kernel today runs servers, desktop PCs, and laptops, as well as being at the heart of the Android OS, which runs the majority of smartphones, tablets, and a multitude of other devices. Device manufacturers increasingly find it important to have support for their products, in the form of *device drivers*, in the Linux kernel. The Linux kernel, however, is fast evolving, with frequent kernel-level API changes. This raises a challenge for device driver developers who have to choose a target kernel version that will be acceptable to the potential users of the device. A solution that helps ensure the continuing availability of the driver code is to target the current mainline version of the Linux kernel, so that the driver code can be integrated into the Linux kernel distribution itself and maintained by the mainline kernel developers [12]. Users, however, typically run older versions of the kernel, which are considered to be more stable. For such users, the driver must then be *backported* to older kernel versions.

Currently backports are typically done manually, on a case by case basis. An alternative is provided by the Linux backports project, which provides a compatibility library to hide differences between the current mainline and a host of older versions, and provides patches that allow a set of 800 drivers to target this compatibility library. These patches are either created manually by the backports project maintainers, or are created using manually written rewrite rules, based on the transformation tool Coccinelle [27]. In either case, however, the backports project maintainer has to determine where changes are needed in the code to backport and how to carry out these changes. Both of these operations are tedious and error prone.

While the Linux backports project provides partial automation, the user is limited to the versions for which backports have been prepared. In this paper, we propose a step towards truly automating these tasks, in the form of a recommendation system for backporting driver files over code changes. Our approach accepts as input a driver file in a given Linux version, the older Linux version to which the driver file needs to be backported (the *target version*), and the git repository that stores the changes to the Linux source code. It first bisects the repository to find two subsequent commits in the repository such that compiling the driver file results in a compilation error in the older commit version and a successful compilation in the newer commit version. Next, our approach analyzes the differences between the two commits and compares them with the line of code containing the error, as indicated by the compiler. Our approach currently only considers cases where there is only one error line and analyzing these differences is enough to fix the error line and backport the driver file. Based on this analysis, our approach constructs a recommendation list that contains possible changes that can be applied to the error line to make the driver compilable in the target version. The changes are ranked by the similarity between the error line and the result of applying the change to the error line. Our experiment shows that, if a semantically correct change exists in the recommendation list, it is often ranked highly.

The contributions of this work are as follows.

1) To the best of our knowledge, we are the first to work on recommending changes with the goal of automating backporting.
2) We propose a recommendation system that identifies a change in the code history that breaks the driver, and recommends code change candidates that enable backporting of the driver code.
3) We evaluate our approach on 100 Linux device driver

32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, North Carolina, October 2-10, 2016.

files. The recommendation list contains the correct backported code for 68 of the device driver files. Among these driver files, 85.3% of the correct recommendations are located in the Top-1 of the recommendation list.

The remainder of this paper is structured as follows. We provide some background in Section II. In Section III, we present our proposed approach. We then describe our experiments in Section IV. We present related work in Section V. Finally, we conclude and mention future work in Section VI.

II. Preliminaries

In this section, we provide some background about the git version control system [8], and about GumTree [4], a tree differencing tool that we use in our approach.

A. Git

Git is a decentralized version control system that has recently become very popular due to the services that it provides and the tools that have been developed around it. Git is currently used by many software projects, including the Linux kernel. Git is designed around a workflow in which developers pull changes from other developers, modify their copy of the code on top of these changes, and request that other developers pull the changes that they have made. Pulling from another repository creates a *merge* node, representing the result of merging the two sets of changes. Technically, the commits are organized as a directed acyclic graph (DAG), although they are often collectively referred to as a *tree*.

To be sure to have access to complete information about earlier changes in a software project, we focus on the case where there is a single main developer of the system, with whom other developers want to regularly synchronize. The development of the mainline Linux kernel follows this model, as developers request that Linus Torvalds pull their changes prior to each release. In this case, we can view the commits and merges made by the main developer as a single primary trunk (level 1), and the commits made by other developers subsequent to pulling from the main developer and prior to requesting a pull from the main developer as being a secondary trunk (level 2), extending from the developer's initial pull to the merge. Such developers may furthermore serve as the main developer with respect to other developers, perhaps their local colleagues, leading to tertiary (level 3), quaternary (level 4), quinary (level 5), senary (level 6) trunks, etc. Figure 1 presents an example git tree illustrating pulls, merges, and primary, secondary, and tertiary trunks.

B. GumTree

A critical point of our approach is to be able to precisely identify changes between two commits in a code base. Line-based tools such as GNU `diff` are not sufficient, because a change can be mixed with irrelevant code fragments that happen to appear on the same line. To address this issue, we use tree differencing, as implemented by the tool GumTree [4]. GumTree identifies common subtrees in an abstract syntax tree, and then integrates common ancestors as long as there are not too many differences among their other descendants. A user study has found that the results of GumTree are considered to be better than those of a text-base differencing tool about half of the time, and mostly the same otherwise.

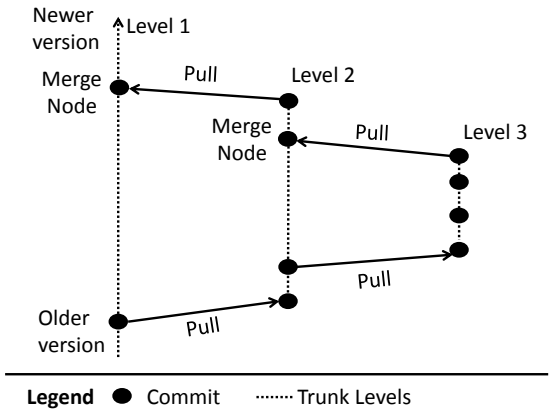

Fig. 1. Example of a Git Tree

III. Proposed Approach

We first present a high-level overview of our automatic recommendation-based backporting approach and then elaborate on the different phases.

A. Overall Framework

As shown in Figure 2, our approach is divided into three phases: 1) error inducing change (EIC) search, 2) code transformation extraction, and 3) recommendation ranking. We define an error-inducing change as a patch between two consecutive commits in which compiling a target backport file in the older commit version leads to a compile error.

In the first phase, our approach gives as input a driver file that needs to be backported (*input driver file*), the Linux version to which we want to backport the driver file (*target Linux version*), and the git repository containing the change history between the target Linux version and the Linux version where the input driver file currently exists (*version control system*). The EIC search phase searches for two consecutive commits such that compiling the input driver file results in a compilation error in the older commit version, and no error in the newer commit version. The goals of this phase are then two-fold: (1) Find the relevant change in the Linux kernel implementation that results in the input driver file not compiling in the target Linux version, (2) Find the changes that have been performed to existing Linux driver files to adapt them to this Linux kernel change. These adaptations are often committed at the same time as the relevant change to the underlying Linux kernel to prevent compilation errors. By reversing these adaptations, we can obtain hints on how to backport the input driver file.

The EIC search phase has one processing component, namely the *EIC Search Engine*, which uses a binary search, starting with the target version, to jump through the change

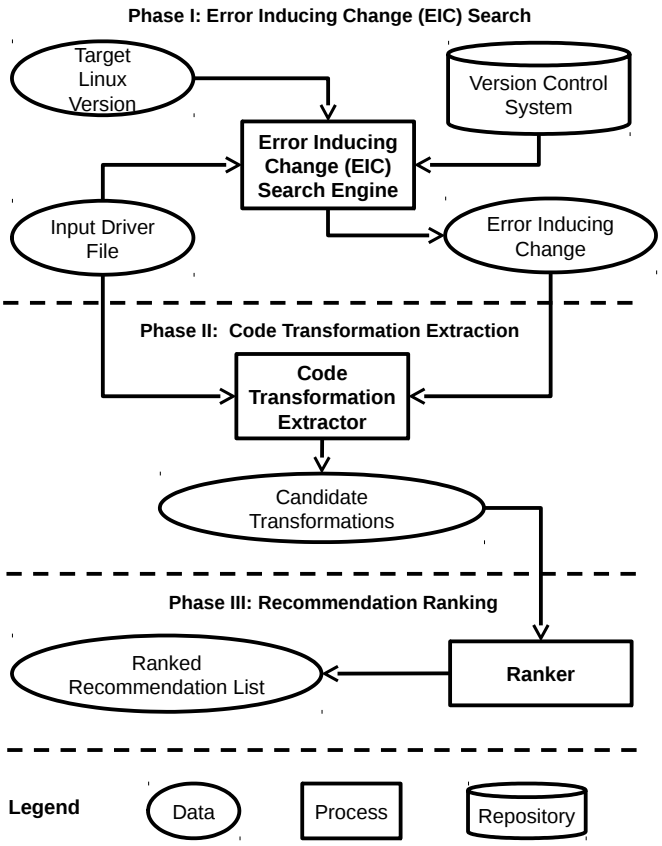

Fig. 2. Our Automatic Backporting Framework

history recorded in the version control system and tries to compile the input driver file in each visited commit version. The search stops when it finds the commit version in which compiling the input driver file leads to no compilation error and the previous commit version still has compilation error. The change between this commit version and the previous commit version is the EIC. For example, an EIC may include the removal of a function definition on which the driver relies. Note that, in a more general case, there would be many EICs, however, we consider only cases where there is only one EIC. This one EIC may consist of many code changes between two consecutive commits that together render the code uncompilable. This EIC is output to the next phase.

In the second phase, our approach takes as input the EIC obtained in the previous phase and the input driver file. This phase searches for changes in the EIC that are relevant to the line in the input driver file that the compiler has marked as erroneous (which we refer to as the *error line*), and generates candidate transformations to backport the input driver file. This phase has one processing component, namely the *Code Transformation Extractor*, which matches the error line with each deleted line in the EIC. It then generates candidate transformations based on how the deleted lines are changed to the corresponding added lines. These candidate transformations are the recommendations produced by our approach.

In the third and final phase, our approach passes the candidate transformations to the *Ranker*, which ranks the transformations based on the similarity between the error line and the result of applying the transformation. We favor the minimal change between them. A developer who needs to backport the driver file can then examine the generated *ranked recommendation list* from top to bottom to find a suitable transformation.

We now describe each phase of our approach in more detail.

B. Error Inducing Change Search

To find the EIC, an approach could be to linearize the git commit history and perform a binary search on it. However, linearizing the history in any way would break the parent-child relationship between two consecutive commits, which represent the actual change made when a developer commits to its local repository. For example, linearizing commit history by date would result in a series of commits that is ordered by date. However, two consecutive commits in this case may not represent the actual change since the two commits might be made by two developers in their own local repository and just accidentally happen at around the same time.

Instead, we follow the approach presented in Algorithm 1. This algorithm takes as input the driver file DF to backport, the original Linux version $origRev$ for which the driver file has been implemented and the target Linux version $targetRev$ to which we want to backport the driver file.

The algorithm starts by retrieving the list $revList$ of all the commit hashes that lie in the main trunk ranging from the original version to the target version (Line 2). To achieve this, we use the command: git log –pretty=%H –first-parent ⟨target version⟩..⟨original version⟩. The option pretty=%H causes the result to be a list of commit hashes and the option –first-parent causes the log to follow only the first parent commit after a merge. Next, at Line 3, the algorithm passes this list to the function EIC_Search, defined just below. EIC_Search first performs a binary search of the commit hashes in $revList$ to find a pair of consecutive commits, $child$ and $errParent$, such that the driver file does compile in the code resulting from $child$ but does not compile in the code resulting from $errParent$. Next, if $child$ is a merge commit (Line 6), then it must have some other parent in which the driver file does compile, because a merge commit does not itself change any code. In Line 7, EIC_Search checks each parent commit other than $errParent$ until it finds one in which the driver file compiles, which is named $okParent$. Based on our assumption that there is a single main developer from whom all code is initially obtained, $errParent$ and $okParent$ have a common ancestor, which is named $ancestor$ in Line 8. This ancestor is obtained using the command: git merge-base $errParent$ $okParent$. EIC_Search then obtains the sequence of commit hashes from $ancestor$ to $okParent$. To achieve this, we use the command: git log –pretty=%H –first-parent ⟨$ancestor$⟩..⟨$okParent$⟩. On the other hand, if the binary search in Line 5 yields a node that commits a

Algorithm 1: Error Inducing Change Search

Input : DF = driver file to be backported
$origRev$= the Linux version where DF works fine
$targetRev$ = target Linux version
Output: error inducing change

1 **Main Algorithm** `Main_Search` ($origRev$, $targetRev$,DF)
2 $revList = [origRev, targetRev]^{FirstParent}$
3 **return** `EIC_Search` ($revList$);
4 **Procedure** `EIC_Search` ($revList$)
5 $(child, errParent)$ = $Binary\_Search(revList)$
6 **if** *child is a merge* **then**
7 $okParent$=*child's parent that can compile*
8 $ancestor = findCommonAncestor(okParent, errParent)$
9 $newRevList = [okParent, ancestor]^{FirstParent}$
10 **return** $EIC\_Search(newRevList)$
11 **else**
12 **return** patch from *child* to the commit before it
13 **end**

code change, rather than a merge node, EIC_Search returns immediately with the patch from the commit node to its previous commit as the result (line 12).

Alternatively to Algorithm 1, we could potentially have used git bisect [9]. In our preliminary experiments, however, we have found that the commit returned by git bisect does not always have the property that the driver file to backport compiles in the returned commit and does not compile in its immediate predecessor. We will study this issue further in the future, but have chosen to rely on our algorithm, which integrates and ensures the property that we require.

Example: An example of the behavior of Algorithm 1 is illustrated in Figure 3. In this example, we take as an input a driver that works fine in the *Original* version. Our goal is to find the error inducing change that helps us backport the driver to the *Target* version, for which the driver currently cannot compile.

We start the search at the level 1 trunk, within the range of the *Original* version and the *Target* version. At this level, we perform binary search to find a pair of consecutive commits *Child L1* and *errParent L1* such that the driver file successfully compiles in the code resulting from *Child L1* but does not compile in the code resulting from *errParent L1*. In this example, since *Child L1* is a merge commit, it must have another parent in a different trunk level, for which the driver file successfully compiles. We denote this parent of *Child L1* as *okParent L2*. Next, we find *Common Ancestor L1*, which is the common ancestor of the *errParent L1* and *okParent L2*. Afterwards, we perform another binary search on the range of commits between *okParent L2* and *Common Ancestor L1* to find a pair of consecutive commits *Child L2* and *errParent L2*, such that the driver compiles with the code resulting from *Child L2* but does not compile with the code resulting from *errParent L2*. We repeat the search in a similar manner until we find a pair of consecutive commits *Child L3* and *errParent L3*, such that *Child L3* is not a merge commit and the driver file compiles successfully in the code resulting from *Child L3*, but does not compile in the code resulting from *errParent L3*.

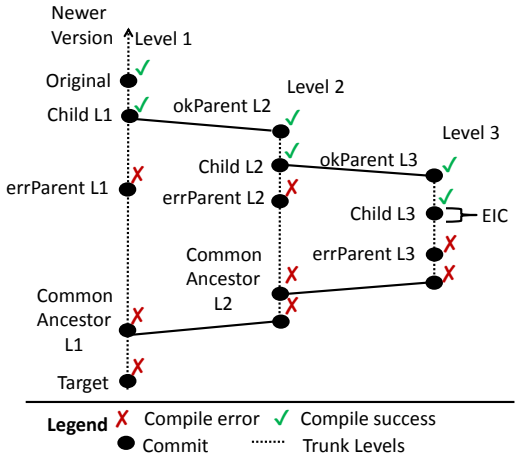

Fig. 3. Example of our Error Inducing Change Search Algorithm

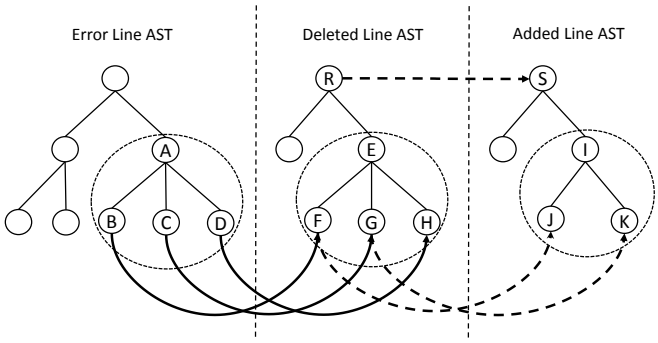

Fig. 4. Code transformation extraction illustration. Error Line AST is extracted from the driver file. Deleted Line and Added Line AST are extracted from historical changes found in the Linux kernel.

Finally, we report *Child L3* as the error inducing change.

C. Code Transformation Extraction

The *Code Transformation Extractor* processes the input driver file and the error inducing change (EIC) obtained by the previous phase. First, we compile the driver file in the target Linux version. By the definition of the backporting problem, we know that this will result in a compilation error. We then record the contents of the line containing the error. The goal is to find a change in the EIC that can be applied to the error line to remove the compiler error. The EIC can be viewed as a code difference (*diff*) composed of *hunks*, each of which is a contiguous sequence of lines corresponding to a sequence of line deletions, line additions, or both. Since we consider a backporting setting (changing code that works on a newer version to adapt it to an older version), to view a transformation in a natural way, we reverse the direction of a normal patch and thus a deleted line is a line in the newer code whereas an added line is a line in the older code. A hunk containing only line additions or deletions does not correspond to a code change from which we can infer a transformation. Thus, we exclude such hunks from our analysis. We also ignore hunks that appear in non source-code files.

Algorithm 2: Code Transformation Extraction

Input : AST_{Err} = AST of the error line
AST_{Del} = AST of a deleted line
AST_{Add} = AST of an added line
Output: A code transformation or $\varnothing$

1 (N_E, M_D) = Tree differencing (AST_{Err}, AST_{Del});
2 (N_D, M_A) = Tree differencing (AST_{Del}, AST_{Add});
3 **if** $N_E \neq \{\}$, $M_D \neq \{\}$, $N_D \neq \{\}$, $M_A \neq \{\}$, *and* $M_D \cap N_D \neq \{\}$ **then**
4 ST_{Err} = Smallest subtree in AST_{Err} covering all mapped nodes;
5 ST_{Del} = Corresponding subtree of ST_{Err} in AST_{Del};
6 ST_{Add} = Corresponding subtree of ST_{Del} in AST_{Add};
7 $varMap$ = Matched identifier mappings from AST^{Err}, AST^{Del}, AST^{Add};
8 ST^{M}_{Add} = Apply $varMap$ to ST^{Add};
9 Output ($ST_{Err} \Rightarrow ST^{M}_{Add}$);
10 **else**
11 Output $\varnothing$;
12 **end**

The *Code Transformation Extractor* constructs an Abstract Syntax Tree (AST) for the error line and for each deleted line and each corresponding added line in the EIC. In general, a line in the source file might not represent a complete term in the C language. Since the complete source files are available, via git, our approach constructs an AST for the function containing each line, and then extracts from that the smallest sub-AST that contains all of the tokens of the considered line. The Code Transformation Extractor then tries to map the nodes of the AST of the error line to nodes of the AST of every deleted line, and the nodes of the AST of a deleted line to the nodes of the AST of every added line in the same hunk. For each combination of error line, deleted line, and added line, the *Code Transformation Extractor* identifies a subtree in the error line's AST that matches a subtree in the deleted line's AST, which in turn matches a subtree in the added line's AST. For each matching, we transplant the matching subtree in the added line's AST to the error line's AST and adapt it with any necessary substitutions of identifier names. The adapted transplantation amounts to a transformation. A set of such possible transformations is output to the next phase.

For each combination of error line, deleted line, and added line, *Code Transformation Extractor* follows Algorithm 2 to produce zero or one candidate transformations. This process involves three main steps: node mapping, extraction of matching subtrees, and subtree transplantation and subterm replacement.

a) Node Mapping: Given an error line's AST, a deleted line's AST, and an added line's AST, we want to find two node mappings: (1) between the nodes in the error line's AST and the deleted line's AST, and (2) between the deleted line's AST and the added line's AST. For this, we use the tree differencing tool GumTree [4] (lines 1-2), described in Section II-B. GumTree maps nodes in the two input ASTs based on some heuristics. It outputs a set of mapped nodes, denoted as $(N, M) = \{(N_1, M_1), ..., (N_k, M_k)\}$ where $N = \{N_1, ..., N_k\}$ and $M = \{M_1, ..., M_k\}$ are the mapped nodes in the two ASTs and k is the number of mapped nodes. If the set of mapped nodes between the error line's AST and deleted line's AST ((N_E, M_D)) and the set of mapped nodes between the deleted line's AST and the added line's AST ((N_D, M_A)) are both non-empty, and the intersection of the mapped nodes in the deleted line's AST based on the two differencing operations is non-empty ($M_D \cap N_D \neq \{\}$), then we proceed with the next steps (Lines 3-9), otherwise we output no transformation (Line 11).

b) Extraction of Matching Subtrees: Given a mapping of nodes in the ASTs, our approach next extracts the minimal subtrees that cover all the mapped nodes. Our approach first identifies the minimal subtree in the error line's AST (Line 4) and the corresponding subtree in the deleted line's AST (Line 5). Next, it finds nodes in the added line's AST that are mapped to the minimal subtree in the deleted line's AST and then extracts a minimal subtree that covers these mapped nodes (Line 6).

As an example, we can employ GumTree to generate the mapping between the error line's AST and deleted line's AST in Figure 4 (solid arrows in the figure). There are three nodes that are mapped between these ASTs: B to F, C to G, and D to H. We then extract the smallest subtree of each tree that covers all of the mapped nodes. This gives the ABCD subtree for the error line's AST and the EFGH subtree for the deleted line's AST.

Next, we again can employ GumTree to generate the mapping between EFGH subtree in the deleted line's AST and the added line's AST in Figure 4 (dashed arrows in the figure). There are three mapped nodes: R is mapped to S, F is mapped to J, and G is mapped to K. Our approach only focuses on the EFGH subtree since it is the only part that matches with the error line's AST and thus, for fixing the error code, our approach only needs to care about the mappings inside this subtree. Next, our approach marks the relevant mapped nodes in the added line's AST and extracts a subtree, IJK, that covers all of the marked nodes. ABCD, EFGH, and IJK are the matching subtrees that are output to the next step.

c) Subtree Transplantation and Subterm Replacement: In this step, we have three matching subtrees, ST_{Err}, ST_{Del}, and ST^{Add}, extracted from the error line's AST, the deleted line's AST, and the added line's AST, respectively. Our approach generates a candidate transformation by transplanting an adapted ST^{Add} to replace ST^{Err} in the error line's AST. Adaptation to ST^{Add} is needed since subterms used in ST^{Add} may differ from those used in ST^{Err}. We infer the necessary replacements of subterms from the mapped nodes between ST^{Err} and ST^{Del}, and between ST^{Del} and ST^{Add}. For a term v in node n in ST^{Err} that is mapped to n' in ST^{Del} (that contains term v'), which is subsequently mapped to n'' in ST^{Add} that contains term v', our approach will store a replacement (v,v') (Line 7). Each replacement (v,v') is applied to nodes in ST_{Add} to create ST^{M}_{Add} where any subterm v' is replaced with v (Line 8). We will then output a transformation ($ST_{Err} \Rightarrow ST^{M}_{Add}$) which corresponds to the transplantation of the adapted subtree (Line 9). At this point, our approach has essentially learned a transformation from an example.

To illustrate how Algorithm 2 works, consider Figure 4. In that figure, we need to transplant the adapted IJK subtree to replace the ABCD subtree in the error line's AST. The IJK subtree is adapted by renaming subterms in the subtree with their corresponding mapped subterms. The subterms are inferred from the mapped nodes between ABCD and EFGH subtrees and EFGH and IJK subtrees. Consider any subterms a and b in the ABCD subtree, any subterms c and d in the EFGH subtree, and any subterm c in the IJK subtree. The mapping may then indicate that a in the ABCD subtree is mapped to c in the EFGH subtree which in turn is mapped to c in IJK subtree. Our approach then creates a replacement (a,c) and will replace all subterms c in IJK with a before transplanting IJK to the error line's AST. The corresponding transformation is (ABCD $\Rightarrow$ IJK[$c \rightarrow a$]).

The above illustration applies to many kinds of changes, such as changes in the field/constant accessed, an argument of a function, the name of a function, or the condition of an if. As a concrete example, we consider a function name change:

Error Line:

```
node=acpi_ns_validate_handle(target_handle);
```

Deleted and Added Lines:

```
- node=acpi_ns_validate_handle(obj_handle);
+ node=acpi_ns_map_handle_to_node(obj_handle);
```

Matching the error line AST against the deleted line AST matches `node` with `node`, `=` with `=`, `acpi_ns_validate_handle` with `acpi_ns_map_handle_to_node`, `(` with `(`, `target_handle` with `object_handle`, and `)` with `)`. All of the tokens are accounted for. In this special case, the subtree transplantation will basically replace the error line with the added line. However, the subterm `object_handle` does not exist in the code containing the error line. Thus, we perform a replacement, (`object_handle`, `target_handle`), which is learned from the mapping. At this point, we have transplanted the added line's subtree to the error line AST and adapted the corresponding subterm to match the one found in the error line.

D. Recommendation Ranking

Ranker takes as input the set of candidate transformations produced by the previous phase and applies each one to the error line in the input driver file to produce a changed error line. Following Occam's razor, our intuition is that the correct transformation is likely to be the one that transforms the error line to something that remains similar to it.

To compute the similarity between the error line and a changed error line, *Ranker* takes the sequence of strings in the error line and changed error line and removes all whitespace from both sequences. It then compares the resulting sequences of characters using Ratcliff and Obershelp's string alignment algorithm [3], which finds a semi-optimal matching of characters in two sequences. Specifically, the algorithm first finds the longest contiguous subsequence between the two sequences. It then recursively processes the subsequences to the left and right of the longest contiguous subsequence. After the matching characters between the two sequences are found, the similarity between the two sequences $SeqSim$ is computed as follows:

$$SeqSim = \frac{2 \times M}{T}$$

where T is the total number of characters in both sequences, and M is the number of matched character pairs. As an example, consider two sequences "abcd" and "bcde". Both sequences have "bcd" sequence as their string subsequence. Thus, T=4 and M=3, and the value of $SeqSim$ is 0.75.

Ranker ranks the transformations by decreasing $SeqSim$ values, and then outputs a *ranked recommendation list*.

TABLE I
DISTRIBUTION OF THE 100 DRIVER FILES IN OUR DATASET ACROSS DEVICE DRIVER FAMILIES

Driver Family	#Drivers	Driver Family	#Drivers	Driver Family	#Drivers
ata	24	char	3	serial	1
media	15	md	3	s390	1
net	11	mtd	3	usb	1
gpu	7	spi	2	power	1
bluetooth	5	hid	2	cpuidle	1
isdn	5	leds	2	ide	1
infiniband	4	scsi	2		
acpi	4	xen	1		

IV. EXPERIMENTS & ANALYSIS

In this section, we first describe our dataset, evaluation measure, and experimental settings. We then list our research questions and provide our experimental results. We finish with a discussion and a description of the threats to validity.

A. Dataset

Our dataset consists of 100 device driver files from Linux 2.6.x versions. Using this dataset, we intend to simulate a backporting scenario. Since general automatic backporting is a new and hard problem, we limit the problem to make it more feasible to solve. Specifically, we select driver files and starting and target versions according to the following criteria.

1) The driver file should have only one changed line of source code between two consecutive Linux versions, e.g., Linux versions 2.6.1 and 2.6.2. We focus on one-line changes to limit the difficulty of making the new code work in the older version. We consider that one-line changes represents a reasonable difficulty for an initial attempt at automatic backporting.
2) Following the first criteria, the driver file will be present in two versions: the old and new versions. We should be able to compile the old and new versions of the driver file in their corresponding Linux kernel versions.
3) When we compile the new version of the driver file within the old version of the Linux kernel, a compilation error should occur. Our goal is to modify the a copy of the new driver to fix this compilation error.

Table I shows the distribution of the 100 Linux driver files selected according to the above criteria.

B. Evaluation Measure

We use Hit@N to measure the effectiveness of our ranking strategy. We define it as follows:

$$Hit@N = \begin{cases} 1, & \text{if } CC \text{ is in the Top-N of } RecCC. \\ 0, & \text{otherwise.} \end{cases}$$

where CC denotes the correct code change, $RecCC$ denotes a code change recommendation list containing the correct code change, and N denotes the number of entries considered at the top of the recommendation list. We compute the average Hit@N across the recommendations to measure the effectiveness of our ranking strategy.

C. Experimental Settings

We simulate a backporting scenario for the 100 driver files in our dataset. For each driver file, we pretend that the driver file is new and has never existed before in the Linux kernel source code. We use the command `make defconfig` to prepare a configuration in which to compile the kernel. We then apply our approach and find the error inducing change for this input driver file. Since the change to the driver file itself is also included in the version control system history, we exclude this change if it appears in the error inducing change. We then find the candidate changes for the input driver file using the remaining examples. To check whether our backport is correct, we simply compare the backported code with the actual old version of the driver file. We consider that the backport is successful only if the backported code is exactly the same as the old version of the code.

D. Research Questions

Research Question 1. *How effective is our proposed approach in extracting correct code changes for a given device driver file?* We report the accuracy of our approach in extracting the correct change. We also investigate the distribution of extracted correct code changes among different driver families.

Research Question 2. *When the correct code change exists in the recommendation list, how high is it ranked by our approach?* We then investigate the rankings of correct code changes in this set. The higher the rankings, the better the recommendations. We measure the recommendation effectiveness using the average Hit@N metric.

Research Question 3. *In what kinds of cases can our approach extract the correct code changes?* We show some cases where our approach extracts correct code changes and assess why our approach works well in these cases.

Research Question 4. *In what kinds of cases is our approach unable to extract the correct code changes?* We show some cases where our approach extracts either incorrect changes or nothing at all. These cases can guide future work.

E. RQ1: Effectiveness of Code Change Extraction

Our approach produces correct code changes to successfully backport 68 out of 100 drivers, giving a success rate of 68%. Table II shows the distribution of the successful cases. Our approach successfully extracts all required code changes for all selected drivers from 7 driver families: ata, bluetooth, mtd, spi, scsi, serial, and cpuidle. For many other remaining driver families, it is successful on some and fails on others.

TABLE II
DISTRIBUTION OF CORRECT CODE CHANGES THAT ARE SUCCESSFULLY EXTRACTED PER DRIVER FAMILY. THE NUMBER IN PARENTHESES IS THE TOTAL NUMBER OF SELECTED DRIVERS IN THE DRIVER FAMILY.

Driver Family	#Successful	Driver Family	#Successful
ata	24 (24)	scsi	2 (2)
media	10 (15)	serial	1 (1)
bluetooth	5 (5)	cpuidle	1 (1)
gpu	4 (7)	md	1 (3)
net	4 (11)	xen	0 (1)
infiniband	3 (4)	s390	0 (1)
isdn	3 (5)	usb	0 (1)
acpi	3 (4)	power	0 (1)
mtd	3 (3)	hid	0 (2)
char	2 (3)	leds	0 (2)
spi	2 (2)	ide	0 (1)

TABLE III
EFFECTIVENESS OF OUR RANKING APPROACH

N	#Correct Code Changes	Average Hit@N
1	50	0.735
2	58	0.853
3	58	0.853
4	58	0.853
5	60	0.882

F. RQ2: Effectiveness of Code Change Ranking

Table III shows the effectiveness of our ranking strategy, for the 68 driver files for which our approach extracts correct code changes. By only recommending the Top-1 code change, we recommend the correct code change for 50 out of the 68 driver files, giving an average Hit@1 of 0.735. Increasing the recommendation to Top-2, we find that there are 8 more driver files whose correct code changes are recommended. This translates to an average Hit@2 of 0.853. Increasing the recommendation to Top-3 and Top-4 does not change the number of driver files for which the correct code change is recommended. When the recommendation is increased to Top-5, there are 2 more such driver files. Thus, by only recommending Top-5 candidate code changes, our approach can successfully find the correct code change 88.2% of the time, thus achieving an average Hit@5 of 0.882.

G. RQ3: Cases where Correct Code Changes are Successfully Extracted

Case 1: Change of record access.

Driver File: drivers/char/drm/drm_agpsupport.c
Error Line:

```
return drm_agp_acquire(
   (struct drm_device*) file_priv->minor->dev);
```

Change example:

```
- struct drm_device *dev = priv->minor->dev;
+ struct drm_device *dev = priv->head->dev;
```

Corrected Error Line:

```
return drm_agp_acquire(
   (struct drm_device*) file_priv->head->dev);
```

In this case, *file_priv→minor→dev* is the part of the error line that provokes a compile error. This code matches *priv→minor→dev* in the deleted line of the change example. This portion of the deleted line matches *priv→head→dev* in the added line. Thus, the *file_priv→minor→dev* portion of the error line is replaced with *priv→head→dev* from the added line. However, we know that the identifier *priv* in the deleted line matches the identifier *file_priv* in the error line. Based on the change example, we also know that *priv* is not changed. Thus, *file_priv* should not change either. We thus map back *priv* to *file_priv*, which is found in the corresponding context in the error line.

Case 2: Deletion of a function argument.

Driver File: drivers/ata/pata_artop.c
Error Line:
```
return ata_pci_sff_init_one(pdev,ppi,&artop_sht,NULL,0);
```
Change Example:
```
- return ata_pci_sff_init_one(dev,ppi,&generic_sht,NULL,0);
+ return ata_pci_sff_init_one(dev,ppi,&generic_sht,NULL);
```
Corrected Error Line:
```
return ata_pci_sff_init_one(pdev,ppi,&artop_sht,NULL);
```

When we match the error line with the deleted line in the change example, the entire *ata_pci_sff_init_one* function call in the former matches the *ata_pci_sff_init_one* function call in the latter. *pdev* of the error line matches *dev* of the deleted line. Similarly, *ppi* is mapped to *ppi*, *&artop_sht* to *&generic_sht*, *NULL* to *NULL*, and 0 to 0. The matched portion of the error line is then replaced with the matched portion of the added line. We then rename the variables in the matched portion of the added line by changing *dev* to *pdev* and *generic_sht* to *artop_sht*.

Case 3: Change of function name.

Driver File: drivers/acpi/acpica/nsnames.c
Error Line:
```
node=acpi_ns_validate_handle(target_handle);
```
Change Example:
```
- node=acpi_ns_validate_handle(obj_handle);
+ node=acpi_ns_map_handle_to_node(obj_handle);
```
Corrected Error Line:
```
node=acpi_ns_map_handle_to_node(target_handle);
```

In the above example, the entire error line has the same structure as the entire deleted line. The function names from the two lines are matched and the argument *target_handle* is matched with the argument *obj_handle*. After the algorithm replaces the error line portion with the added line portion, the function name *acpi_ns_validate_handle* will be changed to *acpi_ns_map_handle_to_node*. To complete the backport, *obj_handle* is renamed to *target_handle*.

Case 4: Change of constant.

Driver File: drivers/media/video/cx23885/cx23885-input.c
Error Line:
```
rc_map=RC_MAP_HAUPPAUGE;
```
Change Example:
```
- dev->init_data.ir_codes = RC_MAP_HAUPPAUGE;
+ dev->init_data.ir_codes = RC_MAP_RC5_HAUPPAUGE_NEW;
```
Corrected Error Line:
```
rc_map=RC_MAP_RC5_HAUPPAUGE_NEW;
```

In this example, again the entire error line matches the entire deleted line, with *rc_map* matched with *dev→init_data.ir_codes*, the assignment node in the error line with the assignment node in the deleted line, and *RC_MAP_HAUPPAUGE* in the error line with the occurrence of the same constant in the deleted line. The error line is then replaced with the added line. To complete the backport, *dev→init_data.ir_codes* is renamed to *rc_map*.

Case 5: Change of if condition.

Driver File: drivers/ata/pata_oldpiix.c
Error Line:
```
if(ata_dma_enabled(adev))
```
Change Example:
```
- if(ata_dma_enabled(adev))
+ if(adev->dma_mode)
```
Corrected Error Line:
```
if(adev->dma_mode)
```

In this case, the if condition in the error line is identical to the one in the deleted line. All matched nodes are of the same type and name. Thus, we directly replace the if condition in the error line with the if condition in the added line to perform the backport.

H. RQ4: Cases where Correct Code Changes are Not Successfully Extracted

Case 1: The correct transformation needs to be learned from multiple deleted-added line pairs.

Driver File: drivers/net/wireless/ath/ath9k/virtual.c
Error Line:
```
txctl.frame_type =
   ps ? ATH9K_IFT_PAUSE : ATH9K_IFT_UNPAUSE;
```
Corrected Error Line:
```
txctl.frame_type =
   ps ? ATH9K_INT_PAUSE : ATH9K_INT_UNPAUSE;
```

In this case, the transformation of the error line into the corrected line involves two changes: transforming *ATH9K_IFT_PAUSE* into *ATH9K_INT_PAUSE*, and transforming *ATH9K_IFT_UNPAUSE* into *ATH9K_INT_UNPAUSE*. Each of these transformations can be obtained from different deleted-added line pairs in the EIC. Although both transformations exist in the EIC, our approach can currently only learn from a single deleted-added line pair. Thus, it applies only one of the two transformations and fixes the error line partially.

Case 2: The EIC provides no relevant example.

The EIC may only contain changes in the definitions that are used by the error line, but all other code that used these definitions could have been updated in earlier commits. Thus, we might not find similar lines of code in the EIC that would suggest how to fix the error line. For such cases, our approach is not be able to recommend correct code changes.

I. Discussion

The quality of our automatic backporting approach depends on the capabilities of each phase in our framework. We assess the possible limitations in each case.

The error inducing change search phase is intended to find a change that contains an example from which we can learn how to backport the code. However, as noted above, all such examples may occur in earlier commits. To address this issue, we would need to extend our search to relevant commits that occurred prior to the error inducing change.

The error transformation extraction phase is intended to learn a potential candidate transformation from a change between a pair of added and deleted lines. Our evaluation shows that there are some cases in which learning from only one pair of added and deleted lines is not enough. Thus, we need a capability to decide which transformation can be combined with another transformation to make a recommendation.

Our approach ranks the candidate transformations, in the ranking recommendation phase, because we have no way to know for sure which transformation is the correct one. A possible way to check the transformation's correctness would be through the use of test cases. Nevertheless, device drivers are difficult to test automatically, because doing so requires installing the device driver on an OS connected to a real hardware or an emulator, and checking whether the hardware is detected and that all of its functionalities can be accessed by the OS. And even if test cases existed for the new driver, they might not be directly usable in the older version. Thus, we would need to to backport the test cases and check whether their backport is correct, repeating the same problem.

Regarding the transformation results, although some may look simple, it is not trivial for developers to perform them. Many people have contributed to the Linux kernel, and around 70,000 commits were made on the Linux kernel in 2015 alone. A developer is unlikely to be knowledgable about the large number of diverse changes made by others. Moreover, state-of-the-art works in automated bug fixing also can only fix a small number of bugs that span one or a few lines of code [16], [19], showing the level of difficulty of automatically inferring correct fix for even one line of code. Although bug fixing and backporting are two different problems, both involve transforming a broken piece of code to another that works.

Last but not least, our approach currently targets backports that require only a one-line change. We have evaluated it on a simulation where we backport a driver file across two consecutive Linux versions. Analyzing a *diff* of two consecutive Linux versions, for all versions between 2.6.11 to 3.13.3, we found 944 driver files having a one line change that affects the driver functionality, meaning that using the newer version of the driver in the previous version of Linux will not work. Thus, even though one-line change is very limited, it still covers quite a number of potential backporting situations.

Another limitation in our evaluation is that we assume that syntactic correctness equals semantic correctness. Since our simulation is based on actual changes, we believe this assumption holds as we only consider backporting to be successful if the resulting code is exactly the same.

In a more general case, a developer may be called upon to backport a driver that requires more than one line of changes. Indeed, the changes required may involve not only multiple lines, but also multiple files, and may require a deeper understanding of the relationships between them. Our approach could still be helpful if the task can be broken down into individual issues that involve only one error line.

J. Threats to Validity

Possible threats to validity include threats to construct validity, to internal validity, and to external validity.

Threats to Construct Validity. These threats refer to the suitability of our evaluation measures. We make use of accuracy and Hit@N as the effectiveness measures of our approach. These measures have been used in past studies and thus we believe the threats are minimal.

Threats to Internal Validity. These threats correspond to experiment errors. To reduce the likelihood of this threat, we have checked the implementation of our approach multiple times. Still, there may be errors that we missed.

Threats to External Validity. These threats refer to the generalizability of our experimental results. We have only investigated 100 Linux device driver files and backporting cases that involve two successive Linux versions. The effectiveness of our approach beyond these driver files and for two arbitrary Linux versions is not guaranteed to be the same. In the future, we plan to reduce this threat by evaluating our approach on more device driver files and arbitrary pairs of Linux versions.

V. Related Work

We now present related work on identifying and analyzing changes, mining change rules, and automatic program repair. Due to page limitations, our survey is by no means complete.

Identifying and Analyzing Changes. Many studies focus on identifying and analyzing changes between two code versions. For example, Neamtiu et al. [23] use partial abstract syntax tree matching to compare the source code of different C program versions. Horwitz et al. [10] focus on identifying semantic changes in programs. Fluri et al. [5] track the co-evolution of comments and source code over multiple program versions, leveraging *Evolizer* [7] and *ChangeDistiller* [6] for extracting fine-grained source code changes between program versions. Marinescu et al. [20] present *Covrig*, an infrastructure that collects static and dynamic software metrics when running different program versions, to study the co-evolution of source code and test cases. Zaidman et al. [30] examine the co-evolution of source code and tests, inferring the development style employed by a number of Java projects.

Our work differs from those above in that, rather than just identifying and analyzing differences, we automatically suggest transformations that can change code in a newer version to be compatible with an older version. We thus need to identify which changes can help us to learn the correct transformation, as implemented by our error inducing change search phase, and to learn and rank these candidate transformations, as implemented by our code transformation extraction and recommendation ranking phases.

Mining Change Rules. Wu et al. [29] propose an approach to mine method call change rules between two versions of a Java program. They combine call dependency and text similarity analyses to extract method change rules. Similarly, Meng et al. [22] extract method change rules between two versions of a Java program. However, rather that considering only one big change between two versions of a program, they consider all changes between two consecutive commits that are located in the change history between the two versions of the program. This allows them to analyze changes in a finer detail and enables them to mine chains of method changes. These works deal with a forward porting problem, which involves porting function calls to use a newer version of a library in Java programming language. We address a backporting problem, which is the task of porting a new version of some code (in our case, drivers) to work with an older version of a system (in our case, the Linux kernel). Thus, backporting can be seen as reverse forward porting. However, different than existing forward porting approaches, we have found that it is not sufficient to focus only on function calls, as we found the need to also port data structures.

Negara et al. [24] develop a tool for automatically mining *frequent* change rules from fine-grained edits. Different from their work, we focus on finding transformations that can backport drivers and these transformations are often not the frequent ones. Andersen et al. [1], [2] infer safe and concise transformation rules from a collection of transformation examples. We do not require users to provide transformation examples, but instead find them automatically.

Automatic Program Repair. Recently, automated program repair has received considerable attention in the software engineering research community, and several techniques have been proposed [11], [14], [17], [18], [19], [21], [25].

Le Goues et al. propose GenProg, the first search-based automated patch generator [17], [18]. GenProg uses genetic programming to generate a large number of possible patch candidates. A valid patch candidate can be found by iterating through a pool of generated candidates and searching for the one that passes all the test cases of a given regression test suite. On a suite of 105 real-world bugs GenProg was able to create plausible fixes (i.e., fixes that pass all test cases) for 55 of them [17]. However, recently Qi et al. have shown that out of the 55 plausible fixes, only 2 are correct fixes [26].

Kim et al. propose Pattern-based Automatic program Repair (PAR), that applies patterns learned from existing human-written patches to fix bugs [11]. Ten fix templates learned from more than 60,000 human-written patches were created and applied to generate patches for buggy programs. The results of their experiment show that PAR can automatically generate patches that are comparable to human-written patches.

Weimer et al. [28] propose an adaptive search strategy that uses a cost model for search-based automated program repair, to reduce the cost of validating candidate patches. The cost estimation allows the model to suggest an appropriate strategy for evaluating repair candidates, e.g., which candidates should be evaluated first and how the test cases should be prioritized to allow an invalid candidate to be recognized as soon as possible. Their experimental results show that adaptive search finds repairs more quickly than GenProg.

Most recently, Long and Rinard and Le et al. propose approaches that learn from historical bug fixes to guide a GenProg-style automated program repair technique to correctly fix bugs with a higher success rate [16], [19]. Mechtaev et al. propose Angelix that can fix bugs using semantics analysis and program synthesis [21]. Subsequently, Le et al. propose to use syntax-guided synthesis [15] and deductive verification [13] for program repair. However, even state-of-the-art approaches can only fix a small number of bugs whose fixes span one or a few lines of code.

Automatic program repair deals with the problem of fixing general bugs, where a bug is usually considered fixed when a program passes all test cases. Our backporting problem is related to automatic program repair in that we need to fix the error line when we compile a driver in the older Linux version. However, we do not have test cases to evaluate whether the fix is correct. Thus, we cast the problem as fix recommendation (i.e., code change recommendation) rather than a fully automatic fix. The approach used by existing automated program repair techniques also differs from our approach; most of them randomly generate possible fixes and rank these possible fixes based on the number of test cases that pass after each fix is applied. In our case we do not have the test cases and thus we generate and rank changes that fix the driver in a different way.

VI. Conclusion and Future Work

In this paper, we have presented a recommendation system that generates a ranked list of candidate transformations for (semi-)automatic backporting of Linux device drivers. Our approach consists of three phases. First, we search for a change that can give us a clue on how to fix the error when we backport a driver to an older Linux version. Then, we extract code transformation candidates that we may apply to fix the error. Finally, we rank the transformation candidates. Our simulated backporting experiment on 100 Linux device drivers shows that our approach can extract the correct transformation for 68% of the device drivers. Among the device drivers having a correct transformation in the recommendation list, our ranking approach ranks first 85.3% of the correct transformations. We then illustrate some successes and limitations of our approach.

In future work, we plan to improve our search algorithm by extending our search beyond the error inducing change, especially to older changes involving the modified data structure or function in the error inducing change. We plan to extend our code transformation extraction algorithm to be able to learn from many deleted-added line pairs. We also plan to consider a more general backporting scenario, such as multi line and multi file backporting. We also plan to explore possibility of lightweight testing for improving the correctness of the backported device drivers. Finally, we plan to experiment on a bigger dataset to ensure the generalizability of our approach.

REFERENCES

[1] J. Andersen and J. L. Lawall, "Generic patch inference," *Autom. Softw. Eng.*, vol. 17, no. 2, pp. 119–148, 2010.

[2] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S. Khoo, "Semantic patch inference," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 382–385.

[3] P. E. Black, "Ratcliff/Obershelp pattern recognition," *Dictionary of Algorithms and Data Structures*, 2004.

[4] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, (ASE)*, 2014, pp. 313–324.

[5] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[6] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering (TSE)*, vol. 33, no. 11, pp. 725–743, 2007.

[7] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with :Evolizer and ChangeDistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[8] "Git," http://git-scm.com/.

[9] "Git bisect," http://git-scm.com/docs/git-bisect.

[10] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1990.

[11] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 802–811.

[12] G. Kroah-Hartman, "The Linux kernel driver interface (all of your questions answered and then some)," https://www.kernel.org/doc/Documentation/stable_api_nonsense.txt.

[13] X.-B. D. Le, Q. L. Le, D. Lo, and C. L. Goues, "Enhancing automated program repair with deductive verification," in *ICSME*, 2016.

[14] X.-B. D. Le, T.-D. B. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair?" in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 427–437.

[15] X.-B. D. Le, D. Lo, and C. L. Goues, "Empirical study on synthesis engines for semantics-based program repair," in *ICSME*, 2016.

[16] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the 23nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016.

[17] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[18] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 54–72, 2012.

[19] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 298–312.

[20] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 93–104.

[21] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 691–701.

[22] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 353–363.

[23] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *International Workshop on Mining Software Repositories (MSR)*. ACM, 2005, pp. 1–5.

[24] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 803–813.

[25] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 772–781.

[26] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 24–36.

[27] L. R. Rodriguez and J. Lawall, "Increasing automation in the backporting of Linux drivers using Coccinelle," in *11th European Dependable Computing Conference - Dependability in Practice*, 2015.

[28] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 356–366.

[29] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 325–334.

[30] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.