Automated Patch Backporting in Linux (Experience Paper)

Ridwan Shariffdeen[*]
National University of Singapore
Singapore
ridwan@comp.nus.edu.sg

Xiang Gao[*]
National University of Singapore
Singapore
gaoxiang@comp.nus.edu.sg

Gregory J. Duck
National University of Singapore
Singapore
gregory@comp.nus.edu.sg

Shin Hwei Tan[†]
Southern University of Science and Technology
China
tansh3@sustech.edu.cn

Julia Lawall
Inria
France
julia.lawall@inria.fr

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Whenever a bug or vulnerability is detected in the Linux kernel, the kernel developers will endeavour to fix it by introducing a patch into the mainline version of the Linux source tree. However, many users run older "stable" versions of Linux, meaning that the patch may also be "backported" to one or more of these older kernel versions. This process is error-prone and there is usually a long delay in publishing the backported patch.

Based on an empirical study, we show that around 8% of all commits submitted to Linux mainline are backported to older versions, but often more than one month elapses before the backport is available. Hence, we propose a patch backporting technique that can automatically transfer patches from the mainline version of Linux into older stable versions. Our approach first synthesizes a partial transformation rule based on a Linux mainline patch. This can then be generalized by analysing the alignment between the mainline and target versions. The generalized rule is then applied to the target version to produce a backported patch. We have implemented our transformation technique in a tool called FixMorph and evaluate it on 350 Linux mainline patches. FixMorph correctly backports 75.1% of them. Compared to existing techniques, FixMorph improves both the precision and recall in backporting patches. Apart from automation of software maintenance tasks, patch backporting helps in reducing the exposure to known security vulnerabilities which have not been fixed.

1 INTRODUCTION

The Linux kernel is one of the most important software projects in the landscape of systems today. Its core functionality is used by a multitude of devices, ranging from servers to IoT devices. To support users with different feature/stability requirements, multiple versions of the Linux kernel are actively maintained. When introducing a patch[1] to the mainline version, maintainers should backport the patches to old stable versions to keep them up-to-date. The increasing number of devices that depend on the Linux kernel and the rapid rate of evolution of the kernel raises a challenge for maintainers to ensure the continuous availability of the kernel code with the latest patches [39].

Although the mainline version of the Linux kernel shares a common codebase with older versions, they typically diverge over time as different features and fixes are added to the latest branch. As a result, when a bug is patched in the mainline version, the patch is often not directly applicable to another version. Given a patch created for the latest version, backporting involves identifying the correct patch location and adapting the patch to an older version. Backporting is typically done manually by a developer, on a case-by-case basis. The manual process of backporting is error-prone and there is usually a long delay in publishing the backported patch. This becomes critical when we consider security patches.

To understand the importance and challenges of backporting patches, we first conduct an empirical study on the Linux kernel versions spanning 2011-19. We found that (1) 51,663 patches have been backported from the mainline to old versions, representing around 8% of all the commits to the mainline version, and (2) the backporting process typically took more than one month. Moreover, backporting patches is not simple copy and paste, as it may involve changing patch locations, changing namespace (the variable or method names used in different versions), and modifying the code logic and structure. These findings indicate that automatically backporting patches is important but challenging.

Existing program transformation techniques can potentially be applied to automate the backporting process. Automated program transformation [6, 7, 15, 21, 24, 33, 34] infers transformation rules from human-written patches, and then applies the inferred rules to unforeseen codebase. These approaches have been used to fix software bugs (e.g. GetaFix [6] and Phoenix [7]), automate repetitive edits (e.g. Refazer [33] and Lase [24]), etc. However, they have two main limitations: 1) they learn transformation rules from multiple human-written patches, which are not always available in reality; 2) the program transformation techniques for fixing software bugs, such as GetaFix [6], Phoenix [7] and Genesis [21], infer transformations from the patches of different applications, so they can only learn general transformation patterns shared by multiple applications, e.g., inserting null-check, fixing API usage errors. These limitations prevent the above techniques from effectively backporting Linux kernel patches. This is because 1) there is usually only one available patch (the one introduced in the mainline version) and 2) most kernel patches are specific to the kernel and the fix pattern cannot be learned from other projects. Although GenPat [15] and Sydit [23] require only one example, GenPat

[*] Joint first authors
[†] corresponding author
[1] Patch generally refer to a change to source files, i.e. to modify, add and delete lines.

requires a large codebase to provide statistical information on how to generalize example, and SYDIT simply generalizes all identifiers and edit positions which may lead to false positives.

The main challenge of synthesizing transformation rules from human patches lies in inferring a proper generalization. An *under-generalized* transformation rule can lead to false negatives: it cannot generate patches for some locations that should be patched. An *over-generalized* transformation rule produces false positives in incorrect locations. The over-generalization problem becomes more serious when only one human patch is available. Consider the following example patch that fixes an off-by-one error by changing < to <=:

$$\texttt{if}\ (\mathit{chunk\_end} + *\mathit{ch} < \mathit{skb}) \mapsto \\ \texttt{if}\ (\mathit{chunk\_end} + *\mathit{ch} <= \mathit{skb})$$

In general, it is hard to infer whether to 1) generalize the variable, e.g. ch, 2) generalize the dereference operation $*ch$, or 3) generalize the whole left operand of the comparison.

Different from existing program transformation systems [6, 7, 21] that transform patches across different projects, our goal is to transform patches between different versions of the same project. Different versions of the same project share similar expressions, algorithms, namespaces, etc. Our main insight is that the similarities between versions can guide us in synthesizing properly generalized transformation rules. Suppose $\mathtt{v}_{\mathtt{mainline}}$ is the original mainline version targeted by developer patch and $\mathtt{v}_{\mathtt{old}}$ is the old version to be backported. For the above example, we observe that $\mathtt{v}_{\mathtt{mainline}}$ and $\mathtt{v}_{\mathtt{old}}$ use many variables (e.g. *chunk_end* and *skb*), expressions and algorithms identically. In the function affected by the patch, we also observe the following: matched statements:

$$\mathtt{v}_{\mathtt{mainline}} : \texttt{skb\_pull}(\mathit{skb}, *\mathit{ch}) \\ \mathtt{v}_{\mathtt{old}} : \texttt{skb\_pull}(\mathit{skb}, \mathit{sctp\_chunkhdr\_t})$$

Based on the usages of these variables in the statement above, $*ch$ should correspond to *sctp_chunkhdr_t* in the old version. To backport this patch from the mainline to the old version, when synthesizing the transformation rule, this observation can guide us to generalize $*ch$ while keeping other elements concrete.

In this paper, we adopt the transformation rule synthesis strategy of REFAZER [33] and investigate how it can be adapted to meet the needs of backporting patches from the mainline to old stable versions. Specifically, we synthesize a transformation rule from the $\mathtt{v}_{\mathtt{mainline}}$ patch and apply the transformation rule to multiple old versions ($\mathtt{v}_1, \mathtt{v}_2, ...\mathtt{v}_n$). First, based on the single $\mathtt{v}_{\mathtt{mainline}}$ patch, we use a REFAZER-like synthesis strategy to produce a generic transformation rule R on top of a *Domain Specific Language*. Since R is prone to overfit the given patch, we extend the REFAZER-like approach with a notion of *partial program transformation rule* R^p, which allows certain fragments of R^p to be generalized according to the context where R^p is applied. For the above example, the partial transformation rule could be:

$$\texttt{if}(\mathit{chunk\_end} \sim \texttt{true} + *\mathit{ch} \sim \texttt{true} < \mathit{skb} \sim \texttt{true}) \mapsto \\ \texttt{if}(\mathit{chunk\_end} + *\mathit{ch} <= \mathit{skb})$$

where identifiers (e.g. *chunk_end*) and expression ($*ch$) are marked as flexible ($\mathtt{e} \sim \mathtt{true}$ means e is flexible). Second, we determine how to generalize these flexible elements according to the alignment of $\mathtt{v}_{\mathtt{mainline}}$ and $\mathtt{v}_{\mathtt{i}}$. This alignment models the matched code elements in $\mathtt{v}_{\mathtt{mainline}}$ and $\mathtt{v}_{\mathtt{i}}$ with respect to the file, function, expression, namespace, etc. The main insight is that similarities and differences of $\mathtt{v}_{\mathtt{mainline}}$ and $\mathtt{v}_{\mathtt{i}}$ modeled by the alignment can guide us to decide which elements should be generalized. This enables us to find an appropriate generalization according to the target version in an on-demand manner.

We implement our approach in a tool named FIXMORPH—i.e., a tool for (MORPH)ing (FIX)es across Linux versions—and evaluate it in two different scenarios. In the first scenario, we construct a dataset including 350 backported patches from the Linux kernel project. We evaluate the effectiveness of FIXMORPH by comparing the syntactic and semantic equivalence of the automatically backported patches with the developer patches. Our results show that FIXMORPH can successfully backport 75.1% of the patches. In the second scenario, we identify 30 patches tagged with CVEs, committed to the mainline branch to evaluate our approach. In this scenario, FIXMORPH can correct backport 21 of them.

Contributions. This paper makes the following contributions.

- We perform a comprehensive study of the backported commits of Linux kernel spanning over nine years and 46 versions;
- We propose a novel transformation rule synthesis that can produce properly generalized transformation rules and automate the patch backport process;
- We design and implement our idea into a tool called FIXMORPH;
- We evaluate FIXMORPH on 350 mainline patches and show that 75.1% of the patches can be backported. The dataset is available as *supplementary material* with this submission.

2 EMPIRICAL STUDY

We conduct an empirical study of changes in the Linux kernel to better understand the extent and characteristics of patch backporting. Specifically, our study answers the following research questions:

RQ1: How many patches are backported per release? What is the percentage of the backported patches?

RQ2: How long does it take to backport patches?

RQ3: How do developers backport patches? Can they directly copy and paste, or do they need to modify the patches, and if so how do they modify the patches?

In our study, we investigated 46 versions (v3.1 to v5.5) of the Linux kernel covering nine years (2011-2019). In total, we collected 633,860 commits submitted to the mainline version of the kernel and 144,437 commits that backport a mainline patch to an older stable version. We focus on changes made to source code files and exclude commits that change other types of files (e.g., configuration files).

2.1 Percentage of Backported Patches

We analyze the percentage of backported patches (commits) out of all released patches. For each release version $\mathtt{v}_{\mathtt{release}}$, we compute the number of patches introduced in $\mathtt{v}_{\mathtt{release}}$ and cross-reference with patches that were backported to older versions. Figure 1 shows the distribution of the percentage of patches that were backported for each $\mathtt{v}_{\mathtt{release}}$. The distribution ranges from 3.87% - 16.29%, and on average, 8% of patches for a release has been backported to at least one older version. In total, among 633,860 mainline patches in released versions, 51,663 have been backported to older versions. A patch is only backported if it is important [2]. As users rely heavily on the old stable versions (much more than the mainline), backporting all those patches is necessary and critical.

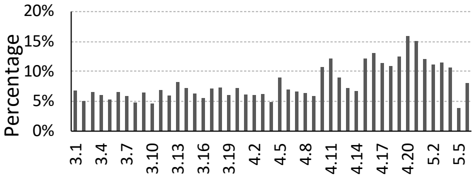

Figure 1: The distribution of backported patches per release

> 51,663 patches, accounting for 8% of all patches (commits), were backported to older versions during 2011-2019.

2.2 How long does it take to backport a patch?

We investigate the delay between the time when a patch is committed to the mainline and when it appears in all relevant stable versions. We measure this delay by computing the difference between the commit date of the patch in the mainline and the commit date of the last backported patch. For example, the patch with commit ID db4175ae[2] was committed on 15 Jul 2014, and it was backported to five stable versions (v3.2, v3.10, v3.12, v3.14 and v3.15). The last backported patch (commit ID 5248ee65) was committed to v3.2.63 on 13 Sep 2014. Hence, the time to backport this patch is 15 Jul 2014 - 13 Sep 2014, which is 60 days.

Figure 2 shows a cumulative distribution function (CDF) for the time of backporting patches in days. For simplicity, we only show the data for a duration of up to a year. 80% of patches took more than 20 days, while around 50% took more than 46 days. We also found that around 10% of backported patches took more than 365 days, amounting to 4844 commits. As some bugs may be security critical, the longer time it takes to backport patches, the higher possibility that such bugs can be exploited by malicious attackers. These results indicate the necessity to accelerate the patch backporting and motivate us to design approaches to automate it.

> Around 50% of backported patches took more than 46 days to be backported from the mainline to old stable versions.

2.3 How does a developer backport patches?

To investigate the patch backporting effort required for a developer, we need to manually inspect the backported patches. We choose to study only the patches backported from a specific version (i.e., v3.8). We label commits based on the difficulty of backporting:

- *Type-I (no changes)*: the backported commit does not require any change from the original patch;
- *Type-II (only patch location changes)*: the patch location(s) (e.g. the containing filename(s) and function name(s), line number(s)) are different in the mainline and the old versions;
- *Type-III (only namespace changes)*: the original patch is adapted by modifying variable names, function names, etc.;

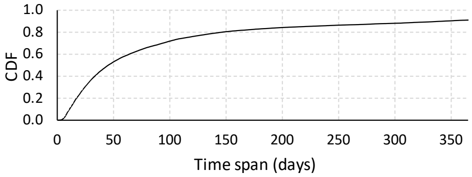

Figure 2: Cumulative distribution of patch backporting time

```
int unix_read (struct
  unix_state *state) {
     ......
+  scm_destroy(...);
}
```

```
int unix_revmsg (struct
  unix_state *state) {
     ......
+  scm_destroy(...);
}
```

(a) Patch location changes when backporting from v4.5 to v3.2

```
- create_seq ("typeinfo",
    0444 , NULL,
    &pageinfo_op );
+ create_seq("typeinfo",
    0400, NULL,
    &pageinfo_op);
```

```
- create ("typeinfo",
    S_IRUGO , NULL,
    &pageinfo_fops );
+ create("typeinfo",
    0400, NULL,
    &pageinfo_fops);
```

(b) Namespace changes when backporting from v5.5 to v3.16

```
if (dev->vendor==ID_INT) {
   ...
+  xhci->quirks |=
     XHCI_AVOID_BEI;
}
```

```
+  if (dev->vendor==ID_INT)  {
+    xhci->quirks |=
       XHCI_AVOID_BEI;
+ }
```

(c) Structure changes when backporting from v4.0 to v3.2

Figure 3: Different types of changes to the original patch

- *Type-IV (patch location & namespace changes)*: the original patch is adapted by changing both the patch location *and* namespace;
- *Type-V (logical or structural change)*: other changes are needed, such as adding extra code or removing irrelevant code.

Table 1 shows the manual analysis results. In our analysis, most of the backporting patches are Type II, which require changes to the patch locations. More than 10% of them are Type-IV or Type-V, which represent the more challenging cases. If a patch has been backported to multiple versions, we noticed that the patches backported to the oldest version are more likely to be Type-IV and V, indicating the challenges of backporting patches to very old versions. According to our manual inspection, the patch location and namespace changes (Type-II, III, IV) are easier to automate, while automating the Type-V changes is more challenging.

Table 1: Developer effort in backporting patches

Label	Description	Count	Percentage
Type-I	no changes	149	22.9%
Type-II	only patch location changes	431	66.3%
Type-III	only namespace changes	0	0%
Type-IV	location & namespace changes	20	3.1%
Type-V	logical and structural changes	50	7.7%

Figure 3 shows three simplified backported patch snippets. Figure 3a shows a backported patch that changes the patch location. This patch was first introduced in the function `unix_read` in v4.5 to fix a memory leak bug, and it was then backported to v3.2, but to a different function `unix_revmsg`. To backport this patch, the developer must manually find the correct function in the target version. Figure 3b shows a backported patch that requires

[2] The detail of each commit can be found in https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux-stable /+/COMMIT_ID^!

```
if (!gcells->cells || skb_cloned(skb)
        || netif_elide_gro(dev)) {
-    return netif_rx(skb);
+    res = netif_rx(skb);
+    goto unlock;
}
```

(a) The patch introduced in v5.1 (commit 2a5ff07a)

```
if (!gcells->cells || skb_cloned(skb) ||
       !(dev->features & NETIF_F_GRO)) {
-    return netif_rx(skb);
+    res = netif_rx(skb);
+    goto unlock;
}
```

(b) Backported patch from v5.1 to v4.9 (commit 7cbb0ab1)

```
if (!cell || skb_cloned(skb) ||
       !(dev->features & NETIF_F_GRO)) {
    netif_rx(skb);
-    return;
+    goto unlock;
}
```

(c) Backported patch from v5.1 to v3.16 (commit 415f08eb)

Figure 4: Sample backporting task

namespace changes. This patch was introduced in v5.5 to fix a vulnerability, and backported to v3.16, by changing the API call from `create_seq` to `create`, and the arguments from `0444` and `pageinfo_op` to `S_IRUGO` and `pageinfo_fops`. Finally, Figure 3c shows a patch that requires structural changes. This patch added a quirk `XHCI_AVOID_BEI` to v4.0 under a if-condition `if (dev->vendor == ID_INTEL)`. However, this if-condition does not exist in Linux v3.2. So, to backport this patch, the developer needs to backport this if-condition as well.

> When backporting patches, a developer needs to find correct patch locations, change the namespace, and modify the program logic and composition of the patch.

3 OVERVIEW

Figure 4 depicts a simplified bug-fixing patch and its corresponding backported patches. This patch was first introduced in v5.1 (Figure 4a) and fixed a fault in the kernel paging request handler. The patch changes an immediate return to a goto to take advantage of the shared error handling code at the end of the function. It also stores the original return value in a variable used by this shared code. This patch was backported to eight stable versions (v3.16, v3.18, v4.4, v4.9, v4.14, v4.19, v4.20, and v5.0). The backported patches for v4.9 and v3.16 are shown in Figures 4b and 4c, respectively. We observe that the if-condition (highlighted in Figures 4b and 4c) of the backported patches is not the same as the if-condition of the mainline version v5.1.

Given the patch p shown in Figure 4a, a developer needs to take the following steps to backport p to older versions. First, the developer needs to analyse p to understand the surrounding context where p is applied and understand how p changes the program. Second, since the mainline version $\mathtt{v}_{\mathtt{mainline}}$ and target version $\mathtt{v}_{\mathtt{i}}$ are not the same, the developer needs to analyze their similarities and differences of $\mathtt{v}_{\mathtt{mainline}}$ and $\mathtt{v}_{\mathtt{i}}$ to find the correct location in $\mathtt{v}_{\mathtt{i}}$ to apply p. At the same time, the developer may need to adjust p according to the context of $\mathtt{v}_{\mathtt{i}}$. Last, the developer produces a patch in the target version. FixMorph tries to automate this process via transformation rule synthesis. Specifically, FixMorph takes the whole if-statement as p's surrounding context and synthesizes a partial transformation rule R^p. The transformation rule is represented using a domain-specific language, which will be explained in Section 4.2. For simplicity, we show R^p for this example as follows:

$$\begin{array}{l}
\mathtt{if}(t_1 \,||\, t_2 \,||\, t_3)\ \{\ \mathtt{return}\ m_1(a_1);\ \} \mapsto \\
\qquad \mathtt{if}(t_1 \,||\, t_2 \,||\, t_3)\ \{\ v_1{=}m_1(a_1);\ \mathtt{goto}\ l_1;\} \\
\ \mathtt{where} t_1.\mathtt{type{=}bool}\ \wedge\ t_1.\mathtt{value}{=}\text{“}!gcells{-}{>}cells\text{”} \\
\qquad \wedge\ t_2.\mathtt{type{=}bool}\ \wedge\ t_2.\mathtt{value}{=}\text{“}skb\_cloned(skb)\text{”} \\
\qquad \wedge\ t_3.\mathtt{type{=}bool}\ \wedge\ t_3.\mathtt{value}{=}\text{“}netif\_elide\_gro(dev)\text{”} \\
\qquad \wedge\ a_1.\mathtt{type{=}struct*}\ \wedge\ a_1.\mathtt{value}{=}\text{“}skb\text{”}\ \wedge\ \ldots
\end{array}$$

The partial transformation rule R^p keeps the keywords (e.g. `if`) and some operators that affect high level transformation structures (e.g. ||) fixed and leaves the other elements in R^p as *flexible* for follow-up adjustment. In this case, the expressions t_1, t_2, t_3, m_1, v_1 and a_1 are marked as *flexible*, meaning that the constraints on them can be relaxed. This way, R^p allows certain expressions to be generalized, so that FixMorph can determine the correct level of generalization according to the target version by relaxing different expressions.

How does FixMorph decide which flexible expressions should actually be relaxed for a given target version? To backport p from v5.1 to v4.9, the original R^p cannot be directly applied. The first two boolean expressions (corresponding to t_1 and t_2 in R^p) are the same in v5.1 and v4.9, but the third expression is different. Therefore, FixMorph relaxes the constraints on t_3 by dropping the constraint on t_3`.value`, allowing it to be a boolean expression with a different value. This leads to the rule, which is used for backporting to v4.9.

$$\begin{array}{l}
\mathtt{if}(t_1 \,||\, t_2 \,||\, t_3)\ \{\ \mathtt{return}\ m_1(a_1);\ \} \mapsto \\
\qquad \mathtt{if}(t_1 \,||\, t_2 \,||\, t_3)\ \{\ v_1{=}m_1(a_1);\ \mathtt{goto}\ l_1;\} \\
\ \mathtt{where} t_1.\mathtt{type{=}bool}\ \wedge\ t_1.\mathtt{value}{=}\text{“}!gcells{-}{>}cells\text{”} \\
\qquad \wedge\ t_2.\mathtt{type{=}bool}\ \wedge\ t_2.\mathtt{value}{=}\text{“}skb\_cloned(skb)\text{”} \\
\qquad \wedge\ t_3.\mathtt{type{=}bool} \\
\qquad \wedge\ a_1.\mathtt{type{=}struct*}\ \wedge\ a_1.\mathtt{value}{=}\text{“}skb\text{”}\ \wedge\ \ldots
\end{array}$$

Backporting to v4.9 required relaxing t_3. Backporting p to v3.16 (see Figure 4c) requires relaxing both t_1 and t_3. Further, backporting the patch to v3.16 requires a post-processing adjustment for the transformation, which will be explained in Section 4.5. We omit the details of the relaxed rule for v3.16.

In this example, FixMorph needs to generate different levels of generalization (by relaxing different flexible expressions) to backport the patch to all versions. The most generalized transformation rule (generalize all flexible expressions $t_1, t_2, \ldots$) is able to transform all the versions (except for the post-processing adjustment). However, it will produce many false positives, i.e., incorrectly transforming some if-statements that should not be transformed, e.g. `if(a || b || c) return foo(i)`.

4 METHODOLOGY

4.1 Preliminaries and Problem Statement

Typed Abstract Syntax Trees. An *Abstract Syntax Tree* (AST) is a tree representation of the syntactic structure of source code. A *typed* AST associates each tree node with one or more *attributes*, including type information (e.g., `int`, `bool`, etc.), text value, filename, function name, etc. We denote the set of typed ASTs as $\mathbb{T}$.

Transformation Rule. A transformation rule $R : \mathbb{T} \rightarrow \mathbb{T}$ formulates how to transform a $\mathbb{T}$ to another $\mathbb{T}$. Rule R can be represented as a pair (`guard`, `transformer`) [25, 33] defined as follows:

- `guard`: $\mathbb{T} \rightarrow$ `Boolean`: `guard` is a conjunction of predicates over AST nodes. Basically, a `guard` tests the type, label and attributes of an AST node and returns a Boolean value representing whether the node satisfies its predicate or not;
- `transformer`: $\mathbb{T} \rightarrow \mathbb{T}$: the `transformer` takes an input $\mathbb{T}$ and constructs another $\mathbb{T}$. It is built from two underlying operations: (1) `select`: returns an existing node from input $\mathbb{T}$ satisfying a given *guard*, and (2) `construct`: returns a new node constructed from a specific node kind, attributes, and children.

Essentially, the rule `guard` determines which AST sub-node should be transformed, and the `transformer` determines how the sub-node should be transformed. Thus, for $t \in \mathbb{T}$, we have $R(t) = \mathtt{transformer}(t)$ when $\mathtt{guard}(t)$ is `true`, otherwise, $R(t)$ is $\perp$.

Transformation Rule Synthesis. Given an input domain $\mathbb{I}$ and an output domain $\mathbb{O}$, *program synthesis* takes a set $\{i_0 \mapsto o_0, ..., i_n \mapsto o_n\}$ of input-output pairs and synthesizes a program $P : \mathbb{I} \rightarrow \mathbb{O}$ such that $P(i_k) = o_k$ for $k \in 0...n$. For this paper, $\mathbb{I}$=$\mathbb{O}$=$\mathbb{T}$, and thus the synthesized program can serve as a *transformation rule* that transforms an input $\mathbb{T}$ to output $\mathbb{T}$. We use the existing program synthesis strategies of Refazer to synthesize transformation rules. In general, the aim is to synthesize a transformation rule (`guard`, `transformer`) that is the generalization of the concrete transformations, so that $\mathtt{guard}(i_k) = \mathit{true}$ and $\mathtt{transformer}(i_k) = o_k$ for all $k \in 0...n$.

Patch Backporting Problem. A patch p can be thought of as a concrete transformation from one $\mathbb{T}$ to another $\mathbb{T}$. Thus, to backport a patch from mainline version $\mathtt{v_{mainline}}$ to old stable versions $\{\mathtt{v_1}, \dots \mathtt{v_n}\}$, FixMorph first uses Refazer's strategies to synthesize a transformation rule $R : \mathbb{T} \rightarrow \mathbb{T}$ using the $\mathtt{v_{mainline}}$ patch p, and then applies R to $\{\mathtt{v_1}, \dots \mathtt{v_n}\}$ to produce patches. Since R is synthesized from a single concrete transformation ($\mathtt{v_{mainline}}$ patch p), we find that R is prone to *overfitting*. That is, R can be applied to $\mathtt{v_{mainline}}$, but often cannot be directly applied to the older versions $\{\mathtt{v_1}, \dots \mathtt{v_n}\}$.

Partial Transformation Rule. To address the overfitting problem, we extend the Refazer synthesis strategy by introducing a notion of *partial transformation rule* R^p. The rule R^p annotates certain predicates as *flexible*. Intuitively, a partial transformation rule R^p is a flexible generalization of the given concrete transformation (i.e. the patch). This flexibility allows R^p to be generalized in an "on-demand" manner according to the context in which R^p is applied. Hence, FixMorph finds an appropriate level of generalization for each old version $\mathtt{v_i}$, allowing backporting to $\mathtt{v_i}$.

4.2 A DSL for Backporting Patches

Refazer performs transformation rule synthesis by searching over a Domain-Specific Language (DSL) for specifying transformation rules. FixMorph extends this DSL to the language $\mathcal{L}_T$ shown in Figure 5 (differences are highlighted in grey) to address the needs of patch backporting. The differences are as follows.

```
rule         := (guard, transformer)
guard        := pred  ~ flexible  | Conjunction(pred, guard)
pred         := IsKind(node, kind)
                | Attribute(node, attr) = value
                | Not(pred)
                | HasType(node, type)
                | InFile(node, fileName)
                | InFunction(node, functionName)
flexible     := true | false
transformer  := select | construct
construct    := Tree(kind, attrs, childrenlist)
childrenlist := EmptyChildren | select | construct
                | Cons(construct, childrenlist)
                | Cons(select, childrenlist)
select       := Match(guard, node)
node         := ...
```

Figure 5: Domain-specific language for transformation rules

First, to allow on-demand generalization, we allow predicates to be marked with a `flexible` annotation, denoted by `pred` $\sim$ `flexible`, where `flexible` is a Boolean value. If a predicate can be relaxed, its corresponding `flexible` annotation will be `true`, otherwise it will be `false`. Second, existing synthesis frameworks focus on the local context (e.g., node kind). However, when backporting patches between different versions, we find that the *global* context (e.g., the file and function name) can also help guide the backporting process. In general, a patch will most likely be backported to a file and function with the same name as in $\mathtt{v_{mainline}}$. To support this feature, we add two predicates `InFile` and `InFunction` to our DSL $\mathcal{L}_T$. Finally, since FixMorph is built on top of typed ASTs, we also add a type checking (`HasType`) predicate to $\mathcal{L}_T$.

4.3 Transformation Rule Synthesis

In this section, we describe how to synthesize a partial transformation rule R^p from a given $\mathtt{v_{mainline}}$ patch p. Given patch p, FixMorph first builds two ASTs $t_i, t_o \in \mathbb{T}$ representing the code before/after the application of p. Essentially, p is represented as an AST transformation $t_i \overset{p}{\mapsto} t_o$. A typical patch p will only affect some subsets of the complete code, e.g., some specific lines, statements, or functions. Rather than representing p as a *global* transformation over the entire file (or files), we restrict t_i and t_o to the *local* AST nodes changed by p as well as some surrounding contexts. Our approach is analogous to the *context diff formats* supported by the standard `diff` and `patch` tools, where the patch p includes not only the changed lines, but also some surrounding unchanged lines for context. The context serves as a reference point and allows for the patch to be applied even if other unrelated parts of the code have been modified. Since we aim to backport patches to older versions with other modifications applied, our motivation is similar. For the context, FixMorph takes the parent and all siblings of any AST node changed by p. For example, the patch shown in Figure 4a changes a branch of an if-statement. FixMorph, therefore, takes its parent node, i.e., the if-statement, as the surrounding context. Thus, the patch p is represented as an AST transformation over the if-statement rather than specific changed nodes. In addition to the

AST context, FixMorph also includes other forms of context in the guard, such as the file and function name of the patch location.

Algorithm. Given an input-output pair (t_i, t_o) extracted from patch p, we first use the synthesis strategies of Refazer to synthesize a transformation rule in the form $(R_{\texttt{guard}}, R_{\texttt{transformer}})$, which is specified using the $\mathcal{L}_T$ DSL in Figure 5. In particular, the synthesis engine first synthesizes the most specific $R_{\texttt{guard}}$ that satisfies $R_{\texttt{guard}}(t_i) = \mathit{true}$. This is essentially a conjunction of all $\mathcal{L}_T$ predicates satisfied by t_i. Similarly, the synthesis engine synthesizes a $R_{\texttt{transformer}}$ that implements the transformation $t_i \mapsto t_o$. However, the transformation rule produced by the synthesis strategies of Refazer is prone to overfit the given input-output pair (t_i, t_o), and may not be generalized to others. Therefore, instead of directly using $R_{\texttt{guard}}$ and $R_{\texttt{transformer}}$, FixMorph marks one or more predicates used by them as being flexible. Specifically, FixMorph marks the predicates of $\mathsf{R}_{\texttt{guard}}$ as flexible to allow relaxing the requirements in finding locations to apply the patch. FixMorph marks the predicates of guard used by select operators as flexible to allow relaxing the requirements in selecting nodes from t_i.

Example 4.1. Consider the following transformation:

$$\begin{array}{l} \texttt{if}(\mathit{chunk\_end} + *\mathit{ch} < \mathit{skb})\ \{...\} \mapsto \\ \qquad \texttt{if}(\mathit{chunk\_end} + *\mathit{ch} <= \mathit{skb})\ \{...\} \end{array}$$

The corresponding partial transformation rule is:

$$\begin{array}{l} \texttt{Tree}(\texttt{IfStatement}, [], [\\ \qquad \texttt{Tree}(\texttt{BooleanExpression}, [], [\\ \qquad\quad \mathit{select}_1, \texttt{Tree}(\texttt{Opcode}, [\texttt{"<="}], []), \mathit{select}_2])\ ]) \end{array}$$

where select_1 is specified by guard:

$$\begin{array}{l} \texttt{HasType}(\texttt{node}, \texttt{Integer}) \sim \mathit{false} \wedge \\ \texttt{IsKind}(\texttt{node}, \texttt{BinaryOperator}) \sim \mathit{true} \wedge \\ \texttt{IsKind}(\texttt{node.kids[2]}, \texttt{DeReferExpr}) \sim \mathit{true} \wedge \\ \texttt{IsKind}(\texttt{node.kids[2].kids[1]}, \texttt{Identifer}) \sim \mathit{true} \wedge \\ \texttt{Attribute}(\texttt{node.kids[2].kids[1]}, \texttt{Text})\texttt{="ch"} \sim \mathit{true} \wedge \dots \end{array}$$

and select_2 is specified by the guard

$$\begin{array}{l} \texttt{HasType}(\texttt{node}, \texttt{Integer}) \sim \mathit{false} \wedge \\ \texttt{Attribute}(\texttt{node}, \texttt{Text})\texttt{="skb"} \sim\ \mathit{true} \end{array}$$

Flexible predicates allow *select* operations to be relaxed. For example, a relaxed select_1 allows for a different "Text" to be used.

By default, FixMorph marks predicates over the "Text" (only leaf nodes have a Text value.), "FunctionName", "FileName", and "Kind" attributes as flexible, and predicates over "Type" as non-flexible. The intuition is that predicates over node types determine the high-level structure of the transformation, and are more likely to be preserved over different versions of the same code.

Remark. *A predicate that is marked as flexible is not necessarily relaxed by the synthesis process. Relaxing all flexible predicates will produce an over-generalized transformation rule, which may produce false positives. For instance, an over-generalized* R^p *from Example 4.1 may incorrectly transform an unrelated node, e.g.,* $\texttt{if}(\texttt{a} + \texttt{b} < \texttt{c})\{\dots\}$.

4.4 Relaxing a Transformation Rule

Once a partial transformation rule R^p is synthesized for $\texttt{v}_{\texttt{mainline}}$ patch p, FixMorph decides how to relax R^p for each old stable version $\{\texttt{v}_1, \dots \texttt{v}_\texttt{n}\}$. To help with this process, we introduce the notion of *alignment* between different versions.

Alignment. We define an *alignment* to be a set of mappings between the code elements or context of $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$. For example, given the following expressions:

$$\begin{array}{ll} \texttt{skb\_pull}(\mathit{skb}, *\mathit{ch}) & \text{from } \texttt{v}_{\texttt{mainline}} \\ \texttt{skb\_pull}(\mathit{skb}, \mathit{sctp\_chunkhdr\_t}) & \text{from } \texttt{v}_\texttt{i} \end{array}$$

an alignment of $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$ would be $\{\mathit{skb} \mapsto \mathit{skb}, *\mathit{ch} \mapsto \mathit{sctp\_chunkhdr\_t}, \dots\}$. For the alignment FixMorph builds multiple mappings, including:

- File: maps of the files between $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$;
- Function: maps of functions between matched file pairs;
- Expression: maps of the matched expressions, e.g., $*\mathit{ch} \mapsto \mathit{sctp\_chunkhdr\_t}$;
- NameSpace: maps of the matched identifiers.

First, FixMorph aligns the source files from $\texttt{v}_{\texttt{mainline}}$ to $\texttt{v}_\texttt{i}$ using a combination of the Git version control history and clone detection. For each modified file, FixMorph uses git to determine the name of the corresponding file in the target version. If git produces no information, FixMorph uses clone detection [16] to find the file in the target version that is most similar to the modified file in the mainline. Next, FixMorph aligns each function, expression, and namespace using a combination of GumTree [8] and anti-unification [29]. Given two ASTs t_1 and t_2, GumTree can generate an *edit script* comprised of insert, delete, move and update operations that can transform t_1 into t_2. Besides, GumTree also constructs a set of matched pairs for the unchanged code elements. For our application, we re-purpose GumTree to generate mappings between two ASTs rather than generate an edit script. Specifically, the GumTree update operation can be used to to derive a set of maps between the code elements (e.g. member accesses, variables) between $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$.

Using GumTree, the mappings between the same kinds of code elements (e.g., identifier to identifier or, assignment to another assignment), can be extracted. We then use an approach based on anti-unification [29] to generate other kinds of mappings such as expression to identifier (e.g., $*\mathit{ch}$ to $\mathit{sctp\_chunkhdr\_t}$). To do so, we analyze the alignment between the matched non-leaf pairs of $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$ via anti-unification. Given ASTs t_i and t_o, their anti-unification is given by $(\tau, \langle \sigma_1, \sigma_2 \rangle)$, where τ is an AST with labelled holes $\{h_0, \dots, h_n\}$, and two substitutions $\sigma_1, \sigma_2 : \{h_0, \dots, h_n\} \rightarrow \texttt{nodes}$ such that $\sigma_1(\tau) = t_i \wedge \sigma_2(\tau) = t_o$. We then use the substitutions to generate a mapping $\sigma_1^{-1}\sigma_2$ between the nodes of t_i and t_o. The mappings produced by GumTree and anti-unification are combined to produce the complete mapping.

Example 4.2. Given the following if-statements:

$$\begin{array}{l} \texttt{if}(\mathit{chunk\_end} + *\mathit{ch} < \mathit{skb})\ \{...\} \\ \texttt{if}(\mathit{chunk\_end} + \mathit{sctp\_chunkhdr\_r} < \mathit{skb})\ \{...\} \end{array}$$

We can apply anti-unification to these ASTs to generate: $(\texttt{if}(\mathit{chunk\_end} + h_1 < \mathit{skb}), \langle h_1 \mapsto *\mathit{ch}, h_1 \mapsto \mathit{sctp\_chunkhdr\_r} \rangle)$.

The anti-unification result is then used to derive the desired mapping $\{*ch \mapsto sctp\_chunkhdr\_r\}$.

Relaxation. Once FixMorph generates a map $\{\mathtt{node}^{\mathtt{i}}_{1} \mapsto \mathtt{node}^{\mathtt{o}}_{1}, \ldots, \mathtt{node}^{\mathtt{i}}_{\mathtt{m}} \mapsto \mathtt{node}^{\mathtt{o}}_{\mathtt{m}}\}$ between $\mathtt{v}_{\mathtt{mainline}}$ and $\mathtt{v}_{\mathtt{i}}$, FixMorph relaxes R^p as follows. For a flexible predicate `pred(node, property)`, meaning the predicate on the `property` (e.g. "`Type`", "`Kind`", "`Text`", etc) of `node`, if and only if the property of `node` is different from the property of its mapped node, FixMorph relaxes `pred` and all the predicates on `node`'s children.

Example 4.3. Let us revisit Examples 4.1 and 4.2. For the predicate `IsKind(node.kids[2], DeReferExpr)` in $select_1$, its corresponding node from $\mathtt{v}_{\mathtt{mainline}}$ is $*ch$, while the mapped node from $\mathtt{v}_{3.5}$ is *sctp_chunkhdr_r*. Since the `Kind` of $*ch$ is different from the `Kind` of its mapped node, FixMorph relaxes this flexible predicate. Besides, FixMorph relaxes the predicate on $*ch$'s child nodes, including:

`IsKind(node.kids[2].kids[1], Identifer)` and
`Attribute(node.kids[2].kids[1], Text)="`*ch*`"`

With the relaxed $select_1$, the transformation rule can generate the transformation:

$$\mathtt{if}(chunk\_end + sctp\_chunkhdr\_r < skb)\ \{\ldots\} \ \mapsto \ \mathtt{if}(chunk\_end + sctp\_chunkhdr\_r <= skb)\ \{\ldots\}$$

4.5 Applying the Transformation Rule

Applying the learnt rule to $\mathtt{v}_{\mathtt{i}}$ itself may not be adequate to successfully transform the program. Although FixMorph learns the transformation rule, the transformed AST could still be incomplete. To make it complete, FixMorph may make a set of post-processing changes, as articulated in the following:

Add missing dependencies. The backported patch p_i may depend on some variables, functions, arguments, etc. that are missing in $\mathtt{v}_{\mathtt{i}}$. FixMorph detects such missing dependencies used by p_i and rectifies them by importing such dependencies. Specifically, FixMorph analyses the AST nodes that are referenced by p_i to find references to missing variables, functions, macros, etc. FixMorph then recursively adds the missing definitions (such as a function or variable declaration, header, etc) to $\mathtt{v}_{\mathtt{i}}$.

Prune irrelevant transformation. Pruning of irrelevant transformations may be required to apply the transformation rule to version $\mathtt{v}_{\mathtt{i}}$. The commit that introduced patch p to $v_{\mathtt{mainline}}$ may include some version-specific changes that cannot be backported to $\mathtt{v}_{\mathtt{i}}$. For instance, the commit in $\mathtt{v}_{\mathtt{mainline}}$ may move a code statement from one location to another location, whereby the code statement does not exist in $\mathtt{v}_{\mathtt{i}}$. FixMorph detects transformations that should be pruned by treating the changes introduced by patch p separately. If FixMorph fails to find an alignment in $\mathtt{v}_{\mathtt{i}}$ for a modified statement, FixMorph prunes the corresponding transformation.

Patch Validation. FixMorph applies R with the above-mentioned post-processing adjustments, to backport the patch p from $\mathtt{v}_{\mathtt{mainline}}$ to $\mathtt{v}_{\mathtt{i}}$. FixMorph first validates the patched $\mathtt{v}_{\mathtt{i}}$ via compilation to check for build errors. If tests are available, FixMorph can further validate the patched $\mathtt{v}_{\mathtt{i}}$.

5 IMPLEMENTATION

Although we synthesize transformation rules using a Refazer-like approach, we cannot directly reuse the Refazer tool since it is designed for C# and Python programs. FixMorph is composed of three main components (*Build engine, Transformation rule synthesis* and *Source code transformation*) and amounts to 10918 code lines in Python and 2645 code lines in C++.

The *Build engine* is used initially to build typed ASTs and finally to validate the patched code. The build engine is based on LLVM/-Clang, to benefit from its facilities for source-to-source transformation and handling of macros. Clang does, however, elide `#ifdefs`, which can lead to missing some code. To limit the number of cases that are considered, our build engine tries two strategies 1) rewrite all `#ifdefs` to `#if 1` and 2) rewrite all `#ifdefs` to `#if 0`.

Transformation Rule Synthesis. To synthesize transformation rules, we used Clang to implement a Refazer-like synthesis engine for the extended DSL. To generate alignment, we use the LLVM GumTree implementation as the AST differencing algorithm [4]. The ASTs used by the original LLVM GumTree implementation only include `Node Kind` and `TextValue`; we added information about types, function names, filenames, etc.

Source code Transformation. While our synthesis algorithm is expressed in terms of ASTs, FixMorph transforms source code directly using LLVM/Clang. Accordingly, the code layout and comments not affected by the patch are preserved.

6 EVALUATION

In this section, we evaluate the effectiveness of FixMorph in backporting patches and answer the following research questions:

RQ1 How effective is FixMorph in backporting patches?
RQ2 How does FixMorph compare with existing tools?
RQ3 Can FixMorph backport fixes of security vulnerabilities?

Dataset: To evaluate FixMorph, we build our dataset in the form of patch pairs ($\mathtt{p}_{\mathtt{mainline}}$, $\mathtt{p}_{\mathtt{i}}$), where $\mathtt{p}_{\mathtt{mainline}}$ is the patch committed to the mainline version, and $\mathtt{p}_{\mathtt{i}}$ is the patch backported to $\mathtt{v}_{\mathtt{i}}$. We build our dataset according to the following criteria:

- Patch $\mathtt{p}_{\mathtt{mainline}}$ was submitted to the mainline during 2011-2019 and versions below 5.0;
- To generate typed ASTs, the mainline version should be compilable before and after introducing $\mathtt{p}_{\mathtt{mainline}}$, version $\mathtt{v}_{\mathtt{i}}$ should be compilable before $\mathtt{p}_{\mathtt{i}}$, and we omit the subjects for which we cannot generate complete ASTs;
- Our prototype only supports modification to *.c files, not header files, hence the patch should only modify *.c files. Further to reduce the complexity, we select patches affecting single *.c file.
- If $\mathtt{p}_{\mathtt{mainline}}$ has been backported to multiple versions, we select the oldest one as $\mathtt{p}_{\mathtt{i}}$ which represents the most challenging task;
- We eliminate the patches that have been used in our study (Section 2.3) to ensure no overlap between our study and evaluation.

Selecting patches affecting only a single *.c file may indeed select simpler patches. Nevertheless, we find that 80% of all backported patches in the Linux kernel (42036/51663) affect only a single file. With the above criteria, we collect 350 pairs to construct our dataset. Table 2 shows the distribution of the patch size (number of lines changed) in our dataset.

Table 2: Patch size distribution in our dataset

1-2	3-4	5-6	7-8	9-10	Total
165(47%)	78(22%)	50(14.5%)	29(8.5%)	28(8%)	350(100%)

Table 3: Effectiveness in backporting kernel patches (a different dataset from the dataset used in Section 2.3)

Type	Total	Plausible	Syntactic	Semantic
I	1	1 (100%)	1 (100%)	1 (100%)
II	235	216 (91.9%)	204 (86.8%)	204 (86.8%)
III	9	7 (77.8%)	4 (44.4%)	7 (77.8%)
IV	30	22 (73.3%)	16 (53.3%)	19 (63.3%)
V	75	41 (54.7%)	22 (29.3%)	32 (42.7%)
Total	350	285 (81.4%)	245 (70.0%)	263 (75.1%)

Moreover, we evaluate FixMorph in backporting bug fixes by selecting 30 patches that fix CVEs using the same criteria. We focus on the CVEs reported during 2014-2019.

All experiments are conducted on a Dell Power Edge R530 with Intel(R) Xeon(R)CPU E5-2660 processor and 64GB RAM.

6.1 [RQ1] Effectiveness of FixMorph

To evaluate the effectiveness of FixMorph, for each pair ($p_{mainline}$, p_i), we use FixMorph to automatically backport $p_{mainline}$ from mainline to v_i, and use the developer backported patch p_i to verify the correctness of the auto-backported patch. We evaluate the correctness of the auto-backported patches by checking their syntactic and semantic equivalence with the developer backported patches.

Table 3 summarizes our evaluation results. Column "Type" indicates the class of subjects as defined in Section 2.3 and "Total" is the number of pairs for each type. Column "Plausible" shows the number of backported patches that can be compiled in the form of x (y%), where x is the total number of instances that were backported and y represents the percentage. Column "Syntactic" and "Semantic" represent the number of patches that are syntactically and semantically equivalent to the developer backported patch, respectively. Out of 350 subjects, FixMorph can backport 285 of them without introducing build failures, which accounts for 81.4%. 245 subjects (70.0%) result in code that is identical to the developer's patch, while 263 subjects (75.1%) result in code that is semantically equivalent to the developer's backported patch. FixMorph shows good results in Type-I, II, and III, indicating its effectiveness in identifying correct patch locations and changing the namespace. Type-IV requires changing both the patch location and namespace, which is more challenging, but FixMorph still can correctly backport 54.7% of them. Type-V includes the most challenging cases, where 42.7% of the patches are correct. The main reason is that FixMorph failed to transform some complex logic and structural changes. To backport, reasoning about the semantics of those patches is needed, which is out of the scope of this work.

6.2 [RQ2] Comparison with Existing Tools

To compare FixMorph with existing techniques, we consider the following baseline approaches:

- `patch`: the `patch` tool [3] from GNU Diffutils; by default, `patch` requires the change to occur at the indicated line numbers;
- $\texttt{patch}^c$: the `patch` tool in context mode [3] (`--context` option), providing flexibility about how many of the patch's context lines are required to be matched;
- Sydit*: our reimplementation of Sydit [23] for C; Sydit is a program transformation tool for Java that learns a transformation rule from a single example, in which it simply generalizes all the identifiers and patch locations. Sydit* follows Sydit, but uses GumTree [8] instead of ChangeDistiller [9] as the AST differencing algorithm. This should benefit Sydit* because GumTree has been shown to be more accurate than ChangeDistiller.

Table 4: Quantitative comparison with existing tools

Type	Total	patch	$\texttt{patch}^c$	Sydit*	FixMorph
I	1	1 (100%)	1 (100%)	0 (0%)	1 (100%)
II	235	124 (53%)	182 (77%)	89 (38%)	204 (87%)
III	9	0 (0%)	0 (0%)	2 (22%)	7 (78%)
IV	30	0 (0%)	0 (0%)	6 (20%)	19 (63%)
V	75	0 (0%)	0 (0%)	0 (0%)	32 (43%)
Total	350	125 (36%)	183 (52%)	97 (28%)	263 (75%)

For a fair comparison, we provide the correct file to patch to all these tools by querying the Git version control system.

Table 4 summarizes our quantitative comparison results. Columns 3–6 represent the correctly backported patches by `patch`, $\texttt{patch}^c$, Sydit*, and FixMorph, respectively. The result for each tool and each class is shown in the form x (y%), where x is the number of patches that have been correctly backported, and y is the accuracy. Despite having the extra advantage of localizing the correct source file, the `patch` tool still failed to correctly backport around half of the instances in Type-II. This illustrates the difficulty in identifying the correct patch locations. In contrast, $\texttt{patch}^c$ performs better than the `patch` tool because it uses context information to find the correct patch location. The key insight we draw from this observation is that *context information is important in identifying patch locations.* Sydit* performs quite well in backporting patches to the correct location due to the usage of additional AST context information. However, since transformation rules synthesized by Sydit* are usually over-generalized, Sydit* incorrectly backports the patches to many locations where the patch should not be applied. We regard a backport as a *false positive* if it produces a patch that modifies the wrong locations in v_i. Overall, Sydit* produces 97 correct patches that are semantically equivalent to the developer patches. FixMorph outperforms all the above tools, especially for the challenging cases, i.e., Type-III, IV, and V. The transformation guided by the alignment of the mainline and target version allows FixMorph to correctly backport more Type-III, IV, and V patches.

To better understand the reliability of each tool, we further evaluate the quality of the transformations for each tool by calculating the precision and recall. Table 5 shows the qualitative comparison results. Columns "P%" and "R%" indicate the precision and recall, respectively. Overall, FixMorph produces much fewer incorrectly backported patches (higher precision) and missed much fewer cases that should be patched (higher recall) than the other tools.

Table 5: Qualitative comparison with existing tools

Type	patch		patch^c		Sydit*		FixMorph	
	P%	R%	P%	R%	P%	R%	P%	R%
I	100	100	100	100	0	0	100	100
II	77	63	99	78	46	69	95	91
III	0	0	0	0	29	50	100	78
IV	0	0	0	0	38	30	86	70
V	0	0	0	0	0	0	78	48
Total	71	42	82	59	44	43	92	80

6.3 [RQ3] Backporting Vulnerability Fixes

To investigate the usefulness of FixMorph in backporting security vulnerability fixes, we evaluate FixMorph on 30 CVE fixes. Table 6 shows the statistics of our targeted CVEs, including the CVE id, vulnerability type, the patch commit id, and the release and target versions. It also shows the evaluation results, where the column“Result” indicates whether the backported patch is semantically equivalent to the developer backported patch.

FixMorph was able to successfully backport 21 out of 30 CVE patches fixing a variety of bugs with semantic equivalence to the developer ported patch. These results suggest that FixMorph can be useful in helping developers fix security vulnerabilities effectively. We also manually analyzed the reasons for the failed cases. For some cases (*e.g.*, CVE-2018-10879), FixMorph could not determine the correct patch locations because the mainline and target version are very different. For some cases, the adaptation requires complex code changes that would involve the understanding of the patch semantics.

6.4 Discussion

In this section, we present some real examples from our evaluation to understand the capabilities of FixMorph.

Better than the developer patch. Backporting patches is an error-prone task, in which developers can make mistakes. In our evaluation, we find that FixMorph performs better than the developer in some cases. For instance, the mainline patch at commit ID 45d73860[3]fixes a bug in a usb driver, in which a delay is added to the function `musb_h_tx_flush_fifo`. However, the developer backported this patch at commit ID 98b91bfa to a different function `musb_host_tx` in the same file although the original function `musb_h_tx_flush_fifo` exists in the old version. This patch was eventually reverted at commit ID c8443922 after 42 days. In contrast, FixMorph transforms the patch at the correct location.

Semantically equivalent with the developer's patch. We find instances where FixMorph backports patch in a syntactically different but semantically equivalent way as the developer. The mainline patch at commit ID 74717b28 fixes a bug in the RTC interface. This patch changes an if-condition by inserting a function call to `ktime_before`. Unfortunately, this function does not exist in v3.2. When this patch was backported to v3.2 (commit ID 69328181), the developer improvised and replaced the function call with a semantically equivalent expression. In contrast, FixMorph imports the missing function `ktime_before` from the mainline to v3.2, resulting in a different, but semantically equivalent patch.

Table 6: Results of backporting CVE tagged bug fixes

CVE ID	Vuln Type	Patch Commit	Release Version	Target Version	Result
CVE-2018-1118	IL	670ae9ca	4.17	4.9	✓
CVE-2018-19985	MO	5146f95d	4.20	3.16	✗
CVE-2019-3701	DoS	0aaa8137	5.0	3.16	✓
CVE-2017-0786	IL	17df6453	4.14	3.16	✓
CVE-2018-1092	NPD	8e4b5eae	4.16	3.2	✓
CVE-2018-1108	RNW	dc12baac	4.17	4.14	✗
CVE-2014-8481	NPD	a430c916	3.18	3.17	✓
CVE-2015-7513	DZ	0185604c	4.4	3.2	✓
CVE-2018-16658	IL	e4f3aa2e	4.19	3.16	✓
CVE-2018-1094	NPD	a45403b5	4.16	4.14	✗
CVE-2018-9363	IO	7992c188	4.18	3.16	✓
CVE-2018-10881	MO	6e8ab72a	4.17	3.16	✓
CVE-2018-10879	UAE	5369a762	4.17	3.16	✗
CVE-2016-9191	DoS	93362fa4	4.10	3.12	✓
CVE-2018-10880	DoS	8cdb5240	4.17	3.16	✗
CVE-2016-0728	IO	23567fd0	4.4	3.10	✓
CVE-2018-11412	MO	117166ef	4.17	3.16	✓
CVE-2017-7184	MO	677e806d	4.11	3.2	✓
CVE-2015-5257	NPD	cbb4be65	4.3	3.2	✗
CVE-2017-12153	NPD	e785fa0a	4.14	3.2	✓
CVE-2016-0758	IO	23c8a812	4.6	3.12	✓
CVE-2016-6213	DoS	296990de	4.12	4.1	✓
CVE-2014-9529	MO	a3a87844	3.19	3.2	✓
CVE-2017-11600	MO	7bab0963	4.13	3.2	✓
CVE-2017-12193	NPD	ea678998	4.14	3.16	✗
CVE-2016-3713	IL	9842df62	4.6	4.4	✓
CVE-2017-8824	UAF	67f93df7	4.16	3.2	✓
CVE-2016-8650	MO	f5527fff	4.17	3.16	✓
CVE-2017-2584	IL	129a72a0	4.10	3.10	✗
CVE-2018-14633	MO	18164943	4.19	3.16	✗
Total	-	30	-	-	21

RNW: Random Number Weakness, *NPD*: Null Pointer, *DoS*: Denial of Service, *UAF*: Use After Free, *MO*: Memory Overflow, *IL*: Information Leakage, *IO*: Integer Overflow, *DZ*: Divide by Zero

Incorrectly backported patch. We also find instances where the backported patch produced by FixMorph has a different behavior from the developer patch. The mainline patch at the commit 7809a611 fixes a regression error by inserting an API call `IS_HASWELL` with `dev_priv` as the first argument. The developer who backported the patch to v3.2 in commit 2dd2c68e used a different variable `dev` as the first argument. However, since the variable `dev_priv` exists in v3.2, FixMorph does not change the argument `dev_priv` to `dev`. FixMorph cannot disambiguate such cases, i.e. keep the original argument `dev_priv`, or change it to `dev`.

Internal DSL. FixMorph's DSL allows expressing transformation rules that can be applied in a flexible way (i.e. for each target version, FixMorph can decide whether the flexible predicates should be fixed or relaxed using the synthesis process). FixMorph's DSL is also only for internal use, to reduce the search space by considering a restricted language, rather than for communication with developers like Coccinelle's DSL[31]. Note that the choice of DSL syntax is not a core contribution of this paper.

[3] The detail of each commit can be found in https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux-stable/+/COMMIT_ID

6.5 Threats to Validity

Several threats may affect the validity of our evaluation. First, since the baseline tool Sydit is designed for Java programs, to compare with it, we implemented Sydit* by ourselves. We tried our best to follow Sydit's design, but the differences in implementation details may still affect its results. Second, although FixMorph shows strong efficacy on the evaluated benchmark, it may perform differently on other subjects. To mitigate this problem, we evaluated FixMorph on a fairly large dataset that covers different scenarios. Last, we manually compare the backported patches with developers' patches to verify their correctness. To reduce the potential bias caused by manual analysis, two authors of this paper independently double checked the correctness of generated patches.

Limitations of FixMorph. Our implementation is based on LLVM/-Clang, and thus inherits the limitations of that framework. Since handling all combinations of compilation options is not scalable, when compiling the project, we only consider two sets of compilation options (see Section 5). This strategy works for most cases, but in some cases, it could result in certain un-compiled blocks of code being unavailable to FixMorph, thus leading to incomplete backporting or even failure. To alleviate this limitation, we allow users to specify the values of preprocessor variables according to their working environment.

7 RELATED WORK

Backporting. To help developers backport patches, several approaches have been proposed [30, 40]. Tian et al. [40] proposed an approach to automatically identify bug-fixing patches, that should be backported to old versions. Ray et al. [30] proposed to detect and characterize porting errors to help developers avoid them. In contrast, we directly backport patches and provide patch suggestions for developers. Another line of relevant work is the Backports Project [1], which enables old Linux kernels to run the latest drivers. The Backports Project develops a set of tools to automate the backporting process for Linux drivers [32, 39] to make them compilable with old kernel versions. The Backports Project uses the program matching and transformation tool Coccinelle [18] to allow developers to express backporting transformation in a generic way that is expected to be applicable to many versions. In contrast, FixMorph is fully automated and does not require manually created transformation rules. A prior approach by Thung et al. [39] automatically extracts code transformation rules. However, this approach requires guidance from compilation errors, and it can only transform patches that affect a single line of code.

Program Transformation. Program transformation techniques infer transformation rules from human-written patches and transfer patches to another codebase by applying the inferred rules. Program transformation has been applied to many software maintenance tasks, including automating repetitive code edits [23, 24, 26, 27, 33], intelligent refactoring [10, 25] and fixing software bugs [6, 7, 21]. Those approaches solve problems similar to FixMorph, but there are key differences. Most existing works infer transformation rules from multiple human patches, while FixMorph synthesizes rules from only one patch. Although GenPat [15] and Sydit [23] also rely on only one example, they either require a large codebase to provide statistical information or synthesize rules by simply generalizing all identifiers, which results in many false positives, as also confirmed by our experiments on Linux. In contrast, FixMorph leverages the similarity between Linux kernel versions to synthesize properly generalized transformation rules.

Program Synthesis. Program synthesis has been applied in many domains (e.g., string manipulation [13], fixing vulnerabilities [41], and program transformation [25, 33]). Syntax guided synthesis [5] (SyGuS) unifies synthesis tasks from different domains by specifying both the syntax and semantics of the desired program. SyGuS constructs programs using the given syntax and semantics with reference to a set of input-output examples. Sketching [36, 37] allows programmers to express their insight about an implementation as a partial program. These techniques have been used in many domains but they require detailed specifications given via multiple examples. In our setting, however, only one example is available.

Program Repair. Automated program repair approaches [12] automatically fix software errors. Existing repair systems usually generate a patch space according to predefined templates [11, 17, 20, 38], then search for (or synthesize) the correct patch guided by a correctness specification given via test cases [19, 22, 28]. FixMorph is not concerned with searching for, or generating a patch, but rather adapting a patch that is already available. To patch Linux code, we cannot assume the existence of tests, and thus none of the test-based repair approaches are directly applicable.

8 REFLECTION AND LESSONS LEARNED

We now conclude the paper by presenting some of the lessons learned from backporting patches in Linux kernel.

Prevalence of backporting in Linux. Our empirical study spans nine years of version histories (2011–2019). It shows that the percentage of backported commits is relatively high (8% of studied commits are backported), and it could take some time to backport patches (10% of backported commits were backported after one year). These observations are aligned with the results in an earlier study that analysed eight years of backporting activities in the Linux kernel (2005–2012) [14]. Given the results in our study, we observe that backporting patches still remains a challenge.

Types of backported changes. Our empirical study indicates that backporting patches may involve complex changes beyond copy-and-paste. Although most backporting tasks involve patches in Type-I and Type-II, existing tools such as GNU `patch` that rely on comparing and merging files cannot address many of them (Table 4). In fact, although Type-II patches seem trivial to backport since no adaptation to the code is required, finding the correct location is nontrivial because (1) the order of the code modifications could be different for the backporting (e.g., the affected functions appear in a different order); (2) although it may seem that only the line number is different in the target version, the surrounding context may also be vastly different, and simple find-and-replace cannot locate such places; (3) some of the patch locations cannot be easily found by referring to Git commit history, and either need expert knowledge of the file changes or some other semantic methods to find the correct place to insert the patch.

Importance of context. FixMorph fails to backport some patches of Types III, IV, and V when the surrounding context is drastically different in the target version. This raises the question of whether context information helps in backporting in general. Indeed, we learned from our results in Table 5 that context information is useful in guiding both $\texttt{patch}^c$ and FixMorph in generating more correct patches for most types of backporting.

Generalization of transformation rules. In terms of the level of generalizations, SYDIT is the most abstract among all evaluated approaches because it generalizes the transformation by abstracting the context, whereas GNU patch is the least abstract because it merely compares and merges files at the line level. The level of generalization affects the precision and recall of each technique. For example, since FixMorph over-generalizes more compared to $\texttt{patch}^c$, $\texttt{patch}^c$ has higher precision for Type II backporting than FixMorph. Meanwhile, as under-generalizing could lead to an increase in false negatives, the recall for Type II is higher in FixMorph than the recall for $\texttt{patch}^c$. To correctly backport patches, we learned that it is important for an automated backporting technique to strike a balance in the generalization such that the context is not over-generalized and the transformation is not over-specific.

Backporting security patches. To reduce the exposure to known vulnerabilities, automatically backporting security patches is an important application of FixMorph. Our evaluation shows that FixMorph is effective in backporting security patches (i.e., successfully backported 21/30, that is, 70% of the evaluated CVEs). Although we only evaluate on the Linux kernel, Table 3 shows that FixMorph can backport patches for various types of vulnerabilities. Instead of generating patches from scratch, future research on automated repair of security vulnerabilities could look into deriving fixes by referring to existing patches (i.e., patch transplantation [35]).

Linux kernel as a study subject. We investigated the backporting activities in the Linux kernel because it is a large-scale widely used codebase. The sheer complexity of the patches and the absence of test cases as specification pose additional challenges for patch backporting. Due to the popularity and importance of the Linux kernel, it could be worthwhile for the program repair community to evaluate repair techniques on Linux. We plan to have an open-source release of FixMorph to accelerate this process. A docker image of our tool is currently available upon request.

Reducing Vulnerability Exposure. At a high level, this paper studies a different problem from automated program repair. Instead of trying to generate or search for fixes, it tries to transplant a known fix into other program versions — the automated patch transplantation problem [35]. This work attempts to show the promise of automated transplantation of patches on the Linux kernel which provides a substantial code-base thereby demonstrating the practical promise of such techniques. Apart from automating software maintenance tasks, such patch transplantation is of significant practical value for reducing exposure to security vulnerabilities.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for insightful suggestions. This research is partially supported by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems).

REFERENCES

[1] 2020. Backports project. https://backports.wiki.kernel.org/index.php/Main_Page. Accessed: 2020-12-20.

[2] 2021. Guidelines for backporting Linux patches. https://www.kernel.org/doc/Documentation/process/stable-kernel-rules.rst. Accessed: 2021-01-24.

[3] 2021. Linux Patch Tool. https://man7.org/linux/man-pages/man1/patch.1.html. Accessed: 2020-11-20.

[4] 2021. LLVM GumTree implementation. https://github.com/llvm/llvm-project/main/clang/tools/clang-diff. Accessed: 2021-1-30.

[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*.

[6] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

[7] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

[8] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[9] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.* 33, 11 (Nov. 2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[10] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[11] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[12] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[13] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 317–330.

[14] Jinru Hua. 2014. *A case study of cross-branch porting in Linux Kernel.* Master's thesis. University of Texas - Austin.

[15] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 255–266.

[16] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 96–105. https://doi.org/10.1109/ICSE.2007.30

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[18] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX Annual Technical Conference (USENIX ATC)*. 601–614.

[19] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[20] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 31–42.

[21] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[22] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 298–312.

[23] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 329–342. https://doi.org/10.1145/1993498.1993537

[24] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 502–511.

[25] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[26] Tim Molderez, Reinout Stevens, and Coen De Roover. 2017. Mining change histories for unknown systematic edits. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 248–256.

[27] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 819–830.

[28] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *International Conference on Software Engineering (ICSE)*.

[29] Gordon D Plotkin. 1970. A note on inductive generalization. *Machine intelligence* 5, 1 (1970), 153–163.

[30] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and Characterizing Semantic Inconsistencies in Ported Code. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, Piscataway, NJ, USA, 367–377. https://doi.org/10.1109/ASE.2013.6693095

[31] Luis R. Rodriguez and Julia Lawall. 2015. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In *Proceedings of the 2015 11th European Dependable Computing Conference (EDCC) (EDCC '15)*. IEEE Computer Society, USA, 132–143. https://doi.org/10.1109/EDCC.2015.23

[32] Luis R Rodriguez and Julia Lawall. 2015. Increasing automation in the backporting of Linux drivers using Coccinelle. In *2015 11th European Dependable Computing Conference (EDCC)*. IEEE, 132–143.

[33] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE)*. IEEE Press, 404–415.

[34] Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, Lingxiao Jiang, David Lo, Julia Lawall, and Gilles Muller. 2020. SPINFER: Inferring Semantic Patches for the Linux Kernel. In *USENIX ATC*. 235–248.

[35] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (Jan 2021).

[36] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (POPL)*. 136–148.

[37] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (POPL)*. 281–294.

[38] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 187–198.

[39] Ferdian Thung, Xuan-Bach D Le, David Lo, and Julia Lawall. 2016. Recommending code changes for automatic backporting of Linux device drivers. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 222–232.

[40] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 386–396.

[41] Fang Yu, Ching-Yuan Shueh, Chun-Han Lin, Yu-Fang Chen, Bow-Yaw Wang, and Tevfik Bultan. 2016. Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 189–200.