SPINFER: Inferring Semantic Patches for the Linux Kernel

Lucas Serrano
Sorbonne University/Inria/LIP6

Van-Anh Nguyen
Sorbonne University/Inria/LIP6

Ferdian Thung
School of Information System
Singapore Management University

Lingxiao Jiang
School of Information System
Singapore Management University

David Lo
School of Information System
Singapore Management University

Julia Lawall, Gilles Muller
Inria/Sorbonne University/LIP6

Abstract

In a large software system such as the Linux kernel, there is a continual need for large-scale changes across many source files, triggered by new needs or refined design decisions. In this paper, we propose to ease such changes by suggesting transformation rules to developers, inferred automatically from a collection of examples. Our approach can help automate large-scale changes as well as help understand existing large-scale changes, by highlighting the various cases that the developer who performed the changes has taken into account. We have implemented our approach as a tool, Spinfer. We evaluate Spinfer on a range of challenging large-scale changes from the Linux kernel and obtain rules that achieve 86% precision and 69% recall on average.

1 Introduction

The Linux kernel is present today in all kinds of computing environments, from smartphones to supercomputers, including both the latest hardware and “ancient” systems. This multiplicity of targets with widely varying needs has implied that the code base has grown steadily over the Linux kernel’s 28-year history, reaching 18M lines of code in Linux v5.4 (Nov. 2019). The complexity of designing an operating system kernel and the need to continually take into account new requirements has implied that the design of internal interfaces must be revisited, triggering the need for repetitive changes among the users of these interfaces that may affect an entire subsystem, or even the entire source tree. The size and number of the needed changes can discourage developers from performing them, and can introduce errors. Furthermore, when code is incompletely transformed, the volume of the performed changes can obscure the conditions that later developers who want to complete the change should take into account.

Since 2008, the automatic C code transformation tool Coccinelle [12] has been part of the Linux kernel developer toolbox for automating large-scale changes in kernel code. Coccinelle provides a notion of semantic patch allowing kernel developers to write transformation rules using a patch-like [16] (i.e., diff-like) syntax, enhanced with metavariables to represent common but unspecified subterms and notation for reasoning about control-flow paths. Given a semantic patch, Coccinelle applies the rules automatically across the code base. Today, Coccinelle is widely adopted by the Linux community: semantic patches are part of the Linux kernel source tree, are invokable from the kernel build infrastructure, and are regularly run by Intel’s Linux kernel 0-day build-testing service [10]. Semantic patches have been written by kernel developers to make timers more secure [5], prepare the Linux kernel for the overflow of 32-bit time values in 2038 [6], reduce code size [27], etc.

Kernel developers use Coccinelle by first writing a semantic patch to perform a desired change, then applying it to the code base using Coccinelle and manually checking the resulting patch, and finally submitting the resulting patch for review and integration, according to the standard kernel development process [28]. Semantic patches have also been recognized as providing a precise means of communication about changes; developers include semantic patches in the commit logs of large-scale changes, and maintainers ask for them if they are not present, showing that semantic patches are considered to be valuable in the review and subsequent maintenance process. Still, there remain large-scale changes in the Linux kernel commit history where Coccinelle has not been used. Through discussions with Linux kernel developers we have learned that some know that Coccinelle would be helpful to them but, as they do not use it often, they do not remember the syntax. They also report that this realization often comes after performing a few manual changes. Furthermore, Coccinelle does not help developers understand existing large-scale changes, if no semantic patch is provided.

To better help developers, we propose to *infer semantic patches from existing change examples*, represented by a collection of files from before and after a change has been applied. Semantic patch inference can help developers understand previous changes without looking at hundreds of change instances. Semantic patch inference can also help developers with little Coccinelle knowledge use semantic patches if they are willing to make a few instances of the change manually.

Inferring semantic patches from real Linux kernel code changes, however, raises some challenges. The Linux kernel is written in C, which is a low-level language. Individual functionalities may be implemented as multiple C statements connected by control-flow and data-flow constraints. These constraints must be captured from the change examples and validated in the code to transform. A single kernel interface may expose multiple functions and data structures that can be used in various ways, any or all of which may be affected by a change. Inference must thus be able to cope with multiple change variants. Finally changes that are relevant to many files may be tangled with *noise*, *i.e.*, changes that are specific to a particular file, and thus are not worth automating. Semantic patch inference should be able to ignore such changes.

In this paper, we propose an approach to semantic patch inference that scales to the needs of systems code, such as the Linux kernel. Starting with the intraprocedural control-flow graphs of the original and changed functions found in the examples, our approach iteratively identifies code fragments having a common structure and common control-flow relationships across the examples, and merges them into a rule proposition. During this iterative merging process, rules are split as inconsistencies appear. Our approach is able to infer semantic patches from examples that overlap, that implement a family of transformations, and that may include noise. We have implemented our approach as a tool, *Spinfer*, targeting semantic-patch inference for the Linux kernel. Spinfer-inferred semantic patches can be read and understood, reviewed, and automatically applied to the Linux kernel.

The contributions of this paper are as follows:

- We propose a taxonomy of challenges that must be handled by transformation-rule inference tools and assess recent work and our proposed approach according to this taxonomy.
- We propose an approach to automatic inference of semantic patches that takes control and data-flow, multiple change instances, multiple change variants and unrelated changes into account. We provide an implementation of this approach for C code in the tool Spinfer.
- We evaluate Spinfer on a large set of 80 changes, affecting thousands of files drawn from recent Linux kernel versions, against semantic patches written by a Coccinelle expert. Generated semantic patches achieve on average 86% precision and 69% recall.

The rest of this paper is organized as follows. Section 2 presents some motivation for our work, our taxonomy and the most closely related work. Section 3 presents our approach and its implementation in Spinfer. Section 4 evaluates Spinfer on 80 recent sets of changes to the Linux kernel. Finally, Section 5 presents other related work and Section 6 concludes.

2 Background and Motivation

We first present an example of a large-scale change from the Linux kernel development history. Based on the challenges identified in this example, we then propose a taxonomy of challenges for transformation-rule inference. We then assess related approaches in terms of this taxonomy.

2.1 Motivating example

Linux kernel timers were originally initialized by a multi-step process, involving a call to **`init_timer`** to initialize a timer structure and two assignments to initialize the fields **`function`** and **`data`**, describing what to do when the timer expires. In Linux 2.6.15, released in 2006, **`setup_timer`** was introduced to combine these operations and thus simplify the source code. Elimination of **`init_timer`** got off to a slow start, with 73 changes in 2008 and then little activity until 2014. In 2014-2016 there were 43, 93, and 37 changes per year, respectively. The remaining 267 calls were removed in 2017, when it was also recognized that incremental initialization using **`init_timer`** represents a security hole.[1]

Figure 1 illustrates some instances of the **`init_timer`** change.[2] We examine these instances in terms of the challenges they pose for semantic patch inference.

Control-flow. The change typically involves removing three statements, *i.e.*, the call and the two field initializations. These three statements are typically part of some larger block of code, and thus do not correspond to an independent subtree of an Abstract Syntax Tree (AST). They are not always contiguous either (**`nicstar.c`**).

Data-flow. The change involves keeping the same expression for the **`init_timer`** and **`setup_timer`** first argument, and for the structure used in the **`data`** and **`function`** field initialization.

Multiple variants. While the examples basically perform the same operations, at a detailed level there are a number of variations. For example, the first two examples initialize both the **`function`** and **`data`** fields, but in the opposite order. Semantic-patch inference should be able to proceed in the face of these variants and generate the needed rules. The last example presents yet another variant that does not initialize the **`data`** field at all, retaining the default value of 0. This variant poses a further challenge, as it overlaps with the other variants, in that all of the examples remove an **`init_timer`** call and the initialization of the **`function`** field. Semantic-patch inference has to *carefully order* the resulting rules such that a rule setting the third argument of **`setup_timer`**, representing the timer data, to **`0UL`** does not apply to code where a value for the **`data`** field is specified.

[1] `https://lkml.org/lkml/2017/8/16/817`

[2] Commit b9eaf1872222. All commits cited in this paper are available at `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`

drivers/s390/block/dasd.c

```
- init_timer(&device->timer);
- device->timer.function = dasd_device_timeout;
- device->timer.data = (unsigned long) device;
+ setup_timer(&device->timer, dasd_device_timeout,
+           (unsigned long)device);
```

drivers/atm/nicstar.c

```
- init_timer(&ns_timer);
+ setup_timer(&ns_timer, ns_poll, 0UL);
  ns_timer.expires = jiffies + NS_POLL_PERIOD;
- ns_timer.data = 0UL;
- ns_timer.function = ns_poll;
```

drivers/net/wireless/intersil/hostap/hostap_hw.c

```
- init_timer(&local->passive_scan_timer);
- local->passive_scan_timer.data =
-           (unsigned long) local;
- local->passive_scan_timer.function =
-          hostap_passive_scan;
-
- init_timer(&local->tick_timer);
- local->tick_timer.data = (unsigned long) local;
- local->tick_timer.function = hostap_tick_timer;
+ setup_timer(&local->passive_scan_timer,
+          hostap_passive_scan, (unsigned long)local);
+ setup_timer(&local->tick_timer, hostap_tick_timer,
+           (unsigned long)local);
```

arch/blackfin/kernel/nmi.c

```
- init_timer(&ntimer);
- ntimer.function = nmi_wdt_timer;
+ setup_timer(&ntimer, nmi_wdt_timer, 0UL);
```

Figure 1: Variants of the `init_timer` change

Multiple instances. Another form of variation in the examples is the number of instances of a given change. Most of the examples initialize only one timer, but `hostap_hw.c` initializes two. To concisely describe the overall change, it is better to obtain a rule for a single timer that applies to this code twice, rather than obtaining a rule specific for this case.

Noise. Change examples can also contain extraneous changes from which it is not useful to infer semantic patches. A single patch may, for example, change a function definition and update the uses of the function accordingly. The change to the definition is a one-time operation, so it is not useful to automate. Another possibility is the presence of other changes, such as minor refactorings. While research on application software has found that the latter tangled changes are frequent [9], the Linux kernel documentation requires that developers separate their changes into one change per patch [28], and thus we expect tangled changes to be a minor issue in our setting in practice.

2.2 Taxonomy

Based on the above examples and study of other large-scale changes in the Linux kernel, we have created a taxonomy, shown in Table 1, of challenges for transformation-rule inference. This taxonomy can be used to characterize particular change examples and to compare transformation-rule inference approaches.

Table 1: Taxonomy of challenges for transformation inference

***C*: Control-flow dependencies**
0. *No* control-flow dependencies between changed terms
1. *Intraprocedural dependencies* between changed terms
2. Intraprocedural dependencies between changed and *unchanged terms*
3. Some terms must *not* appear within relevant control-flow paths
4. *Interprocedural* control-flow dependencies between changed and unchanged terms
***D*: Data-flow dependencies**
0. *No* data-flow dependencies between changed terms
1. Data-flow dependencies between changed terms
2. Data-flow dependencies between changed and unchanged terms
***I*: Change instances per function**
0. One instance
1. Multiple instances
2. Overlapping instances
***V*: Change variants**
0. One variant
1. Multiple variants
2. Multiple variants with specific order
***N*: Noise (errors and unrelated changes)**
0. *No* noise
1. Contains noise

The taxonomy considers the relationship between changed terms in a single change instance (C and D), the ways in which multiple change instances can appear within a single function (I), and the possibility of change variants (V) and unrelated changes (N, noise). For control-flow dependencies, the taxonomy entries range from no dependencies (C_0) to changes that requires intraprocedural dependencies to changed terms (C_1) to changes that involve dependencies between both changed and unchanged terms across multiple functions (C_4). For data-flow dependencies, the taxonomy entries range from no dependencies (D_0) to data dependencies between both changed and unchanged terms (D_2). For multiple change instances, the taxonomy entries range from one instance per function (I_0) to overlapping instances (I_2). For multiple variants, the taxonomy entries range from one variant at all change instances (V_0) to multiple variants that have to be considered in a specific order (V_2). For noise, the taxonomy entries have either the absence of noise (N_0) or presence of noise (N_1). Spinfer targets C_1, D_2, I_2, V_2, N_1 (control-flow and data-flow constraints, potentially overlapping change instances, multiple order-dependent change variants, and the possibility of noise in the examples).

2.3 Existing tools

A number of tools have previously been developed to infer transformation rules or update API uses automatically. Most of these tools target user-level applications written in object-oriented languages, typically Java, while we target an operating system kernel written in C. While these different kinds of code bases raise different challenges, all of the issues we identify with existing tools also apply to our setting.

Individual examples. Sydit [17], A4 [11], MEDITOR [30], APPEVOLVE [7], and GENPAT [8] generate transformation rules from individual change examples. All of these approaches except GENPAT abstract over the examples by abstracting over variables. In practice, change examples mix generic computations and computations that are specific to a particular application. Abstracting over variables is not always sufficient to abstract away these application-specific computations. Such approaches thus obtain low recall on anything but the simplest examples. GENPAT [8] bases abstraction decisions on the frequency of various terms in code repositories such as GitHub. As GENPAT's abstraction strategy is based on the properties of a large code corpus rather than on the change itself, it may infer transformations that are too generic, transforming code that should not be changed, thus reducing precision. The evaluation presented in the work on GENPAT reflects these issues.

Approaches that abstract from individual examples may perform incorrect transformations if the examples are not given to the tool in the right order. For example, in the **`init_timer`** example, if **`nmi.c`** is provided as an example first, then all timer initialization code will be incorrectly updated with the data value 0, even if another data value is provided. APPEVOLVE addresses this issue by computing a common core of the provided examples, and attempting to apply the generated rules in order of their distance from this common core. The rule generated from the **`nmi.c`** example, however, is smaller than the others, and would thus be tested first by such a strategy. APPEVOLVE additionally proposes to use testing to validate the changed code, but the Linux kernel does not provide a comprehensive suite of easy-to-run high coverage test cases.

Several of these tools are also not able to identify change application sites. For example, Sydit requires the user to specify the affected function, while APPEVOLVE requires the user to specify the affected file name and line number. Manually specifying change sites is not practical for a code base the size of the Linux kernel.

In terms of the taxonomy, Sydit, A4, MEDITOR and GENPAT target C_0, D_2, I_1, V_2, N_0; APPEVOLVE targets C_0, D_2, I_1, V_2, N_1 but requires preliminary manual rewriting of the change examples [29].

Multiple examples. LASE [18], REFAZER [23], and Spdiff [1, 2] learn from multiple examples, identifying how to abstract over these examples based on the commonalities and differences between them.

Based on a set of change examples, LASE represents modifications as a sequence of AST edits and solves the Largest Common Subsequence Problem to find a transformation rule. This method implies that LASE cannot learn from examples containing unrelated changes or multiple variants, as we have seen for **`init_timer`**. Moreover these edit subsequences do not capture control-flow constraints and thus can generate incorrect changes, as illustrated below. In terms of the taxonomy LASE targets C_0, D_2, I_0, V_0, N_0

REFAZER represents a transformation as a list of rewrite rules in a domain-specific language. It clusters changes from multiple examples and then infers one rule for each cluster. Like LASE this list of rewrite rules does not incorporate control-flow constraints and thus can generate incorrect changes. In terms of the taxonomy REFAZER targets C_0, D_2, I_2, V_1, N_0.

Spdiff abstracts over common changes across the examples, incrementally extending a pattern until obtaining a rule that safely describes the complete change, taking control-flow constraints into account. It targets the Linux kernel and produces Coccinelle semantic patches. In terms of the taxonomy Spdiff targets C_2, D_2, I_2, V_0, N_0, however when tested on all of our examples it produced no correct results. For even simple function call replacements, we found that Spdiff spends much time on finding more complex but actually incorrect solutions.

We can note that, with the exception of Spdiff, all existing tools do not handle control-flow relationships between changed terms. To illustrate the impact of control flow, we have performed an experiment on Linux kernel-like code using LASE[3] (the distributed implementation of Refazer only supports restricted subsets of Python and C#, making it difficult to test in practice). Consider the following Linux kernel change example:[4]

```
- tport = kmalloc(sizeof(struct ti_port), GFP_KERNEL);
+ tport = kzalloc(sizeof(struct ti_port), GFP_KERNEL);
  if (tport == NULL) {
    dev_err(&dev->dev, "%s - out of memory\n", ...);
    status = -ENOMEM;
    goto free_tports;
  }
- memset(tport, 0, sizeof(struct ti_port));
```

Thus, from examples such as this one, LASE infers a rule that makes the following incorrect changes:

```
  if (a)
-   x = kmalloc(sizeof(*x), GFP_KERNEL);
+   x = kzalloc(sizeof(*x), GFP_KERNEL);
  else
-   memset(x, 0, sizeof(*x));
```

[3]LASE targets Java. We use C in the presented examples for consistency with the rest of the paper.

[4]Commit 7ac9da10af7f

and

```
- x = kmalloc(sizeof(*x), GFP_KERNEL);
+ y = kzalloc(sizeof(*y), GFP_KERNEL);
  y = kmalloc(sizeof(*x), GFP_KERNEL);
- memset(y, 0, sizeof(*y));
  memset(x, 0, sizeof(*x));
```

The first case shows that the inferred rule does not check that the **memset** affects the result of the call to **kmalloc** in the same control-flow path, thus performing an incorrect transformation. The second case shows that the inferred rule modifies the wrong **memset**, *i.e.*, the one affecting **y** rather than the one affecting **x**, and, worse, rearranges the variables so that **x** is no longer initialized. LASE only updates the first match in each function, and thus the second call to **kmalloc** is left unchanged. The former example may be unlikely for the specific functions of **kmalloc** and **memset**, but there is an instance analogous to the latter example in the Linux kernel history.[5]

Our approach. Given the difficulty of choosing an appropriate degree of abstraction from only one example, and given the availability of multiple examples of the use of most interfaces in the Linux kernel, our approach relies on multiple examples. Our approach is able to infer and express both control-flow and data-flow relationships that must be respected between fragments of code that are to be transformed. It adapts to variations in the examples to produce multiple rules and tolerates noise. Finally it produces transformation rules in a readable notation that is close to C code and is familiar to Linux kernel developers.

To the best of our knowledge, our approach is the only one capable of handling all these challenges at the same time and requires no rewriting of example code, making it suitable for systems code transformations.

3 Approach and Tool Design

In this section, we first give an overview of our approach, and then present the various steps used by our tool Spinfer to realize this approach. Spinfer is implemented as 10.6K lines of OCaml code. For parsing C code and mining control-flow relationships, it reuses the infrastructure of Coccinelle [12].

3.1 Overview

Our goal is to produce Coccinelle transformation rules based on a set of provided change examples, consisting of pairs of functions from *Before* and *After* the change. To motivate the steps of our approach, we first consider a semantic patch, composed of 4 rules, that a Coccinelle expert might write based on the **init_timer** examples illustrated in Section 2:

```
@@expression T,F,D;@@
- init_timer(&T);
+ setup_timer(&T, F, D);
  ...
- T.data = D;
- T.function = F;
```

```
@@expression T,F,D;@@
- init_timer(&T);
+ setup_timer(&T, F, D);
  ...
- T.function = F;
- T.data = D;
```

```
@@expression T,F,D;@@
- T.function = F;
- T.data = D;
  ...
- init_timer(&T);
+ setup_timer(&T, F, D);
```

```
@@expression T,F;@@
- init_timer(&T);
+ setup_timer(&T, F, 0UL);
  ...
- T.function = F;
```

The semantic patch consists of four transformation rules for the four different patterns of changes that occur in the examples. Each rule initially declares some *metavariables*, representing abstracted subterms, and then describes the changes as a sequence of lines of code patterns annotated with **-** and **+**, indicating code to remove and add, respectively. "**...**" represents a control-flow path, connecting the code matched by the pattern before the "**...**" to the pattern after the "**...**". Coccinelle applies a semantic patch to C source code in terms of the C source code's control-flow graph (CFG). A CFG as used by Coccinelle contains a node for each semicolon-terminated statement in the function, as well as for each if header, while header, etc. Coccinelle matches each rule of a semantic patch against the CFG of each function in a C file. When a match is found, Coccinelle transforms the code found in each matched CFG node according to the **-** and **+** annotations. More details about Coccinelle's semantic patch language SmPL are presented elsewhere [3].

We can observe that these four transformation rules share some constituents:

- Removal of a call to **init_timer**: **init_timer(&T);**
- Removal of a **data** field initialization: **T.data = D;**
- Removal of a **function** field initialization: **T.function = F;**
- Addition of the call **setup_timer(&T, F, D);**

We call these constituents *abstract fragments*, as they are fragments of a semantic change and each of them has a particular role in that change. Identifying a semantic patch as an assembly of small abstract fragments can help to solve the challenges highlighted in the introduction. In this view:

- Control-flow dependencies are the rules to assemble fragments together.
- Data-flow dependencies are the rules to match metavariables together.
- Variants are different assemblies of abstract fragments.
- Errors and noise are very unpopular abstract fragments.

[5] Commit 87755c0b3af6.

One can thus resolve these challenges by identifying the abstract fragments to consider and by determining how to assemble them together.

Following this observation, our approach focuses on finding abstract semantic patch fragments that will be assembled into one or more semantic patch rules. Each rule will match one of the variants illustrated by the examples. We have implemented this approach as a tool named Spinfer.

Starting from a set of *examples*, consisting of pairs of files before and after some changes, Spinfer constructs a set of transformation rules describing the changes. For this, Spinfer first identifies sets of common removed or added terms across the examples, then generalizes the terms in each set into a pattern that matches all of the terms in the set, and finally integrates these patterns into transformation rules that respect both the control-flow and data constraints exhibited by the examples, splitting the rules if necessary when inconsistencies appear.

Spinfer is organized as follows:

1. Identification of abstract fragments: Spinfer first clusters subterms from the examples having a similar structure and generalizes each cluster into an abstract fragment that matches all the terms in the cluster.

2. Assembling the rule-graphs: Spinfer then combines the abstracted fragments into a *semantic patch rule-graph*, a representation of a transformation rule as a graph, where nodes represent fragments to add and remove, and where edges are determined by control-flow dependencies exhibited in the associated examples.

3. Splitting: When assembling fails or when Spinfer detects data-flow inconsistencies, Spinfer splits existing rule-graphs into more specific ones.

4. Rule ordering: Finally, Spinfer orders the generated rules, removing redundant ones, to maximize precision and recall while minimizing the number of rules for the final semantic patch.

3.2 Identification of abstract fragments

The goal is to cluster nodes sharing similar subterms to form abstract fragments. Given that we have no a priori knowledge of the change variants illustrated by our examples, we must make an arbitrary decision about the granularity at which to investigate their subterms. Concretely, we choose the granularity of individual statements in a straightline code sequence, as well as function headers and headers of conditionals and iterators. An example from the `init_timer` code would be `init_timer(&device->timer);`. Such terms have the desirable property of being complete statements and expressions, with simple control-flow relationships between them. We refer to these terms as *nodes* as they will later correspond to the nodes of the control-flow graphs (CFGs) that we use to validate the control-flow constraints.

Spinfer then proceeds with an initial clustering of the CFG-nodes to be removed and added according to the examples. This clustering is independent of control-flow information, and is refined by control-flow constraints in the subsequent step. Each cluster will be represented by the smallest abstraction that matches all members in the cluster, known as the anti-unifier [21,22] of the cluster. Clustering code fragments using anti-unification has already been used in REVISAR [24] and CodeDigger [4]. However these approaches give the same weights to all anti-unifiers, regardless of their popularity, which cannot help to discriminate noise from relevant nodes. Spinfer overcomes this limitation by using techniques from text mining.

Node weighting: To facilitate clustering and noise detection, we want to give higher weight to anti-unifiers that are likely to form correct abstract fragments, and lower weight to anti-unifiers that are either too specific, manifested as rarely occurring across the set of examples, or too generic, manifested as occurring frequently within a single example, to describe the change. This goal is very similar to the goal of the *term frequency – inverse document frequency (TF-IDF)* [26] process used in text mining to highlight words that particularly characterize the meaning of a given document in a corpus. Spinfer requires the inverse notion, i.e., anti-unifiers that are common to many documents (i.e., functions), but do not occur too frequently in any given document.

Concretely, Spinfer uses a process that we call *function frequency – inverse term frequency (FF-ITF)*. In FF-ITF terms are anti-unifiers and a term weight increases with the number of functions that contains nodes matching this term (function frequency), and decreases with the number of nodes matching this term which appear in the same function (inverse term frequency).

The first step is to count how many times each anti-unifier appears. To do so, Spinfer first represents each node as a set of anti-unifiers that can match this node, from very abstract ones, that are likely to be shared by several nodes, to very concrete ones; we only consider anti-unifiers that at least abstract over local variables. Then given an anti-unifier $\mathcal{A}$, a set of functions F and a particular function $f \in F$ that contains the set of nodes N_f, our weight $w_{\mathcal{A},f}$ is:

$$\begin{aligned} \mathrm{FF}_{\mathcal{A}} &= \frac{|\{f' \in F : \mathcal{A} \in f'\}|}{|F|} \\ \mathrm{ITF}_{\mathcal{A},f} &= \log \frac{|N_f|}{|\{n \in N_f : \mathcal{A} \in n\}| + 1} \\ w_{\mathcal{A},f} &= \mathrm{FF}_{\mathcal{A}} \times \mathrm{ITF}_{\mathcal{A},f} \end{aligned}$$

This weight function is closely related to the one used in TF-IDF. We illustrate why this weight function works with an example on the `init_timer(&ntimer);` code fragment. We consider here only the 3 following anti-unifiers: (1) **`Expr;`**, (2) **`init_timer(&ntimer);`** and (3) **`init_timer(&Expr);`**. (1) is too generic and matches every statement, consequently it will have high function frequency but a very low inverse term frequency. Conversely (2) is too specific and probably matches only one node, so it will have a low function frequency but a high inverse term frequency. (3) matches **`init_timer`** function calls and nothing else, making it a good candidate to form correct abstract fragment. It will have both a high function frequency and a high inverse term frequency, which will result in a higher weight than (1) and (2).

Noise detection: The result of the node weighting gives a set of weighted anti-unifiers for every modified node of each example. The next step is to separate noise from relevant nodes. As noise is composed of very unpopular code fragments, all its anti-unifiers are either too generic or have very low function frequency, and as a consequence, have low weights. On the contrary relevant nodes have at least one anti-unifier with a high weight. Thus it is possible to distinguish noise nodes from relevant nodes by looking at the weight of their highest weighted anti-unifier.

To decide if a node is relevant, Spinfer compares the weight of the node's highest weighted anti-unifier to the average of the weights of the highest weighted anti-unifier of each node. If it is below a certain distance from this average, the node is considered to be noise. We have performed a separate experiment on the noise detection and looked for the distance at which we could mark nodes as noise without producing false positives. We found the ideal distance to be slightly less than 3 standard deviations.

Nodes marked as noise are dropped from further processing by Spinfer.

Clustering: In this step, we want to group together nodes that share a common high-weighted anti-unifier. For this Spinfer, proceeds with the clustering of the nodes not identified as noise. Spinfer first assigns to each node a characteristic vector encoding the weights of all anti-unifiers for this node. Our approach uses agglomerative hierarchical clustering with complete linkage, described by Zhao *et al.* [31], on our characteristic vectors. This approach has already been used for document classification in conjunction with TF-IDF weighting [31].

The number of clusters is determined using the best average *Silhouette score* [25], a score estimating the quality of the clustering, for all possible numbers of clusters. After this procedure we obtain groups of nodes that are very similar, and that will be transformed to abstract fragments in the next step.

We illustrate this procedure with the **`init_timer`** change. In each of our **`init_timer`** examples, three nodes are removed and one node is added. The clusters are shown on the left side of Figure 2. Each element of a cluster is annotated with the example from which it comes and its position in that example.

Cluster (code and CFG-Diff node)	Abstraction
`init_timer(&device->timer);` `init_timer(&ns_timer);`	`init_timer(&X0);`
`device->timer.function = dasd_device_timeout;` `ns_timer.function = ns_poll;`	`X0.function = X1;`
`device->timer.data = (unsigned long) device;` `ns_timer.data = 0UL;`	`X0.data = X1;`

Figure 2: *Before* clusters and anti-unifiers from the **`init_timer`** examples. **`X0`** and **`X1`** are metavariables.

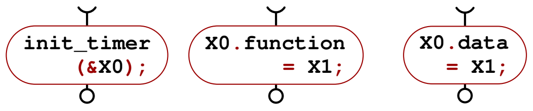

Figure 3: **`init_timer`** *Before* fragments

For each cluster, Spinfer then create its anti-unifier, that retains the common parts of the terms in the cluster, and abstracts the subterms that are not common to all of the terms as *metavariables*, *i.e.*, elements that can match any term. The right side of Figure 2 shows the anti-unifier for each cluster.

Anti-unifiers represent sets of similar terms, but do not provide any control-flow information. In order to prepare for the next step, which constructs a semantic patch rule proposition based on control-flow constraints, Spinfer next expands each anti-unifier into the fragment of a control-flow graph that the anti-unifier's constituent node fragments represent, called an abstract fragment. An abstract fragment is composed of at least one node that contains the abstracted pattern, and a non-empty list of pairs of entry and exit points. Multi-node abstract fragments are created for complex control-flow structures such as conditionals and loops. Figure 3 shows some fragments for our **`init_timer`** example.

3.3 Assembling the semantic patch rule-graph

In order to address the first semantic patch inference challenge of capturing control-flow constraints, Spinfer relies on the notion of *dominance* [15] in CFGs. A node A *dominates* a node B in a directed graph $\mathcal{G}$ if every path from the entry node of $\mathcal{G}$ to B goes through A. A is then a dominator of B. Similarly, a node B *postdominates* a node A if every path from A to an exit node of $\mathcal{G}$ goes through B. B is then a postdominator of A. This notion generalizes straightforwardly to sets of nodes. Dominance characterizes the semantics of Coccinelle's "`...`" operator. A pattern of the form A ... B matches a fragment of C code if the set of terms matching

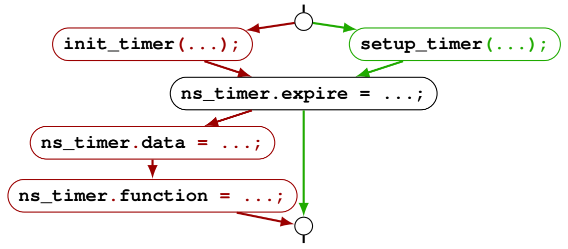

Figure 4: CFG-Diff for the **nicstar.c** **init_timer** change

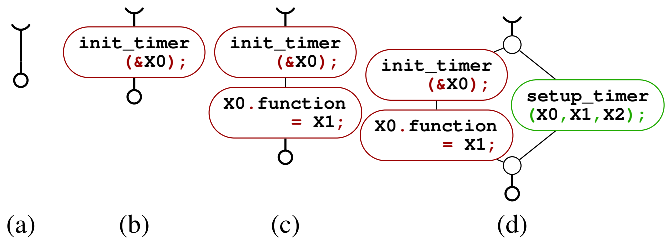

Figure 5: Steps in semantic patch-rule-graph construction

B postdominate the term matching *A*, meaning that from the code matching *A*, all outgoing paths reach code matching *B*.

Spinfer first constructs the control-flow graph (CFG) of each changed function in the examples, before and after the change. Each pair of before and after CFGs is then merged into a single CFG, referred to as a CFG-Diff. In the CFG-Diff, removed nodes from the before CFG are colored *Before* (red/-dark grey) and added nodes from the after CFG are colored *After* (green/light grey). An extract of the CFG-Diff for the **nicstar.c** change is shown in Figure 4.

To construct a semantic patch rule, Spinfer first constructs a *semantic patch rule-graph*, incrementally adding first *Before* fragments and then *After* fragments as long as the dominance relations in the semantic patch rule-graph respect the dominance relations in the example CFGs associated with the fragments.

We present the semantic patch-rule-graph construction algorithm in terms of the **init_timer** example. Spinfer starts with an empty semantic patch rule-graph, consisting of two special nodes, a global entry node and a global exit node, and a directed edge from the global entry node to the global exit node (Figure 5(a), the half-circle is the global entry node and the full circle is the global exit node). By convention, the global entry node dominates every node and every node postdominates it, and the global exit node postdominates every node and every node dominates it.

For our example, out of the available fragments (Figure 3), suppose that Spinfer first chooses the one representing the **init_timer** calls. This fragment can be added straightforwardly (Figure 5(b)) because the start node and the end node dominate and postdominate every node, respectively. Next, Spinfer may add the fragment representing the initialization of the timer's **function** field, below the node representing the **init_timer** call. Indeed, in both *Before* CFGs, the call to **init_timer** dominates the **function** field initialization, either because of a direct connection between them, or because of a control-flow path consisting of straightline code between them. The **function** field initialization likewise postdominates the **init_timer** call. The resulting semantic patch rule-graph is shown in Figure 5(c).

We defer the treatment of the initialization of the **data** field to Section 3.4, and turn to the integration of the *After* fragment, *i.e.*, the abstracted call to **setup_timer**. While the addition of the *Before* fragments relies on dominance relations between the fragment and the nodes already in the semantic patch rule-graph, this is not always possible for *After* nodes. Indeed, there exist no dominance relations between the *Before* and *After* nodes. When an addition is in fact a replacement, we observe that the *After* node and the *Before* node it is replacing share common context (non-modified) predecessors and successors (see Figure 4). Moreover, this context does dominate/postdominate both the deletion and the addition. Spinfer exploits this property to insert *After* fragments. For this, Spinfer replaces each entry and exit pair of the *After* fragment by the nearest context predecessors and successors, if any, and checks the domination properties on these context nodes instead. If the properties hold then the fragment is inserted. The context used is replaced by *Merge* nodes, that are omitted in the final semantic patch.

3.4 Splitting the semantic patch rule-graph

Assembling the semantic patch rule-graph can fail, due to inconsistencies. This situation occurs when there are multiple change variants in the examples, our third semantic patch inference challenge. One possible inconsistency is incompatible domination properties, representing inconsistent control flow. Another is the inability to map metavariables to terms in a way that allows the added code to be constructed from the removed code, representing a form of inconsistent data flow. Spinfer splits the semantic patch rule-graph and reduces the genericity of metavariables to address these issues.

Control-flow inconsistencies. Trying to insert the **data** field initialization fragment into the rule-graph shown in Figure 5(c), reveals a control-flow inconsistency: **dasd.c** initializes the **data** field after the **function** field, while **nicstar.c** initializes it before. To proceed, Spinfer chooses an element of the **data** field initialization cluster, say the one from **dasd.c**, and splits the cluster into one cluster of the instances that respect the same control-flow constraints and another cluster for the remaining instances. The latter is deferred for later integration. At the same time, Spinfer splits the semantic patch-rule graph into one copy derived from the graphs whose **data** field initialization respects the same control flow constraints as the **dasd.c** initialization and another copy for the rest. The latter rule-graph copy is likewise pushed onto a stack for later treatment.

After splitting the cluster and the semantic patch rule-graph, Spinfer can add the fragment representing `data` field initializations that are consistent with `dasd.c` into the rule-graph below the `function` field initialization, and then proceed to the integration of the *After* fragment, as described previously. With this rule-graph complete, Spinfer then proceeds to the next rule-graph on the stack and the remaining fragments.

A semantic patch rule-graph split may lead to a situation, as we have here, where a rule-graph represents only a single example. In this case, Spinfer leaves the generated semantic patch rule abstract, according to the metavariables motivated by the clustering, to allow the generated semantic patch rule to match code from other files. This extra abstraction, however, may lead to false positives, if the rule is so generic that it matches parts of the code that should not be transformed. In a final step, Spinfer validates the complete generated semantic patch on the complete example files, which may contain a great deal of code other than that of the functions the rule was learned from. If this validation shows that a rule causes false positives, Spinfer specializes the rule according to the specific code fragments that motivated its construction, resulting in a safer, but potentially less widely applicable, semantic patch rule.

Data-flow inconsistencies. A semantic patch rule needs to be able to construct the *After* code from the information found in the *Before* code, to carry out the intended change. Thus, in a completed semantic patch rule-graph, Spinfer reassigns the metavariables, `X0`, `X1`, etc. (see Figure 2), that were local to each fragment, in a way that is consistent across the proposed semantic patch rule. It may occur that no consistent assignment is possible, and some metavariables may remain in the *After* fragments that are not instantiated by the *Before* fragments.

As an example, suppose we add the fourth example of Figure 1 to our set of fragments. This example has no `data` field initialization, and thus it will cause a split from the others, producing a separate semantic patch rule-graph. This rule-graph will indicate removal of the abstract fragment `init_timer(&X0);` and removal of the subsequent fragment `X0.function = X1;`, and the addition of the fragment `setup_timer(X0,X1,X2);`. `X2`, however, represents `0UL`, which is not represented by any of the metavariables of the removed code. In this situation, Spinfer agglomeratively considers subsets of the examples contributing to the semantic patch rule-graph, to determine whether respecializing the metavariables to the contents of the considered subsets can produce a consistent metavariable assignment. In our case, there is only one associated example, and Spinfer eliminates `X2` entirely, replacing it with `0UL` in the generated semantic patch.

3.5 Rule ordering

Finally Spinfer pretty prints each semantic patch rule-graph into a rule, and then orders the rules to form a single semantic patch. This last step is important because graph splitting can produce rules that subsume other rules or rules that must be executed in a certain order to limit the number of false positives. For instance, for the `init_timer` example, a rule matching the `nmi.c` variant must be executed after all other rules that could have consumed the missing `data` field.

To solve these issues Spinfer first looks for semantic patch rules that subsume other rules. Let R_1 and R_2 be two rules and let TP_1 and TP_2 be the sets of true positives and FP_1 and FP_2 the sets of false positives produced by applying R_1 and R_2 respectively, by using Coccinelle on the provided examples. R_1 is said to be subsumed by R_2 if TP_1 is a subset of TP_2 and FP_1 is a superset of FP_2. When Spinfer detects a rule that is subsumed by others it is eliminated from the final semantic patch. Finally, Spinfer orders the remaining semantic patch rules, by looking at the F2 score (a combination of precision and recall that favors high recall) of tentative semantic patches each consisting of a pair of rules. From these results, it does a topological sort to order the complete set of semantic patch rules into a single semantic patch.

4 Evaluation

In this section, we evaluate Spinfer on two datasets composed of real changes from the Linux kernel. We then illustrate a failure case of Spinfer in which it generates an incorrect semantic patch and explain why it was generated this way. With this example, we illustrate how a developer can easily fix a semantic patch to produce a correct one.

4.1 Methodology

Our first dataset is a collection of 40 sets of changes in the Linux kernel, that have been selected to challenge Spinfer, by focusing on the higher levels of the taxonomy. Consequently, it is not intended to be representative of the real taxonomy distribution of Linux kernel changes. The sets include some changes that have been performed using Coccinelle as well as recurring changes found in recent Linux kernel versions v4.16-v4.19, identified using the tool `patchparse` [20]. We refer to this dataset as the *challenging dataset*.

Our second dataset is composed of 40 sets of changes randomly picked from changes to the Linux kernel in 2018. To build this dataset we first identified 175 large-scale changes (changes from one developer affecting at least 10 files) from the changes performed in 2018 and then we randomly selected 40. We refer to this dataset as the *2018 dataset*.

For each dataset we describe the interesting aspects of its taxonomy and we perform two experiments:

The *synthesis experiment* learns semantic patches from the full dataset containing all the files and evaluates the resulting semantic patch on the same set of files. This experiment evaluates the degree to which Spinfer is able to capture the changes found in the examples provided to it. It is relevant to the user who wants to understand a previously performed change. Without Spinfer, such a user has to read through the entire patch and collect all of the different kinds of changes performed, with no way to validate his understanding. Spinfer does both the inference and validation automatically.

The *transformation experiment*, on the other hand, learns semantic patches from a reduced dataset composed of the first 10 changed files, as indicated by the commit author date (ties between files in the same commit are broken randomly), or half of the full dataset if the full dataset contains fewer than 20 files. This experiment evaluates the resulting semantic patch on the portion of the dataset that does *not* include the set of files from which the rules were learned. This experiment is relevant to the user who wants to change new code by bootstrapping a semantic patch.

We recall that Spinfer targets the whole taxonomy, except in terms of control flow where it targets only C_0 and C_1, but not the higher levels. Since not all examples are in Spinfer's targets in terms of the taxonomy, for each experiment we separate the analysis of the results according to whether the examples are in Spinfer's scope or not.

We use classic metrics that are applied for evaluation of program transformation tools: *precision*, *i.e.*, the percentage of changes obtained by applying the inferred semantic patch that are identical to the expected changes in the examples, and *recall*, *i.e.*, the percentage of expected changes in the examples that are obtained by applying the inferred semantic patch. To address the issue of *noise* we compare the changes performed by the semantic patch generated by Spinfer against those performed by a human-written semantic patch created by a Coccinelle expert. In this way, we benefit from the intuition of the human expert to filter out noise. As there are many ways to write a semantic patch for the same change, we do not directly compare the syntax of the generated semantic patch against the human-written one, and instead focus on the results of applying the semantic patch itself.

4.2 Experiments on the *challenging dataset*

The taxonomy of the changes for the *challenging dataset* is shown in Figure 6. The majority of transformations in this dataset require taking into account both control-flow and data-flow relationships, with 21 examples requiring at least control-flow dependencies on unchanged terms (C_2), which is outside of the scope of Spinfer (C_0 or C_1). Three quarters of the transformations have multiple variants including one quarter with variants that need to be performed in a specific order, features that are targeted by Spinfer.

To evaluate Spinfer in terms of the correctness of the transformation. We look at the ratio of correctly modified CFG-nodes, either successfully deleted, or added to the correct location in the CFG of the example. As Spinfer prints around one line of semantic patch per modified node, this roughly translates to the ratio of correct lines in the generated semantic patch. Figure 7 gives the recall and precision for the node modifications produced by the generated semantic patches, with each bar corresponding to one set of changes. The lighter color of the bars on the right indicates that the examples are not in Spinfer target in terms of the taxonomy. Missing values for the transformation experiment at indices 1, 2 and 12 correspond to sets of changes with 2 modified files or fewer. In this case, performing the transformation experiment is not possible as Spinfer needs to learn from at least two files. Other missing values indicate that Spinfer either did not generate anything, or generated a semantic patch that was not applicable to the evaluation dataset. These cases count as having both a recall and a precision of 0.

On average, the semantic patches generated by Spinfer achieve 87% precision and 81% recall for the synthesis experiment, and 86% precision and 49% recall for the transformation experiment. For the part of the examples in Spinfer's scope in terms of the taxonomy, Spinfer obtained 88% precision and 91% recall for the synthesis experiment, and 93% precision and 62% recall for the transformation experiment. Precision and recall are much lower for examples outside Spinfer's scope, averaging 86% precision and 72% recall in

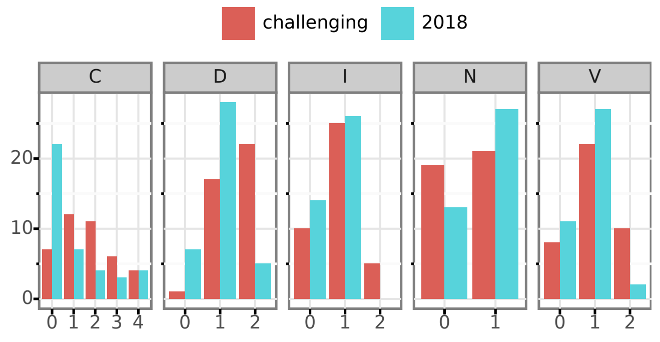

Figure 6: Taxonomy of the two datasets

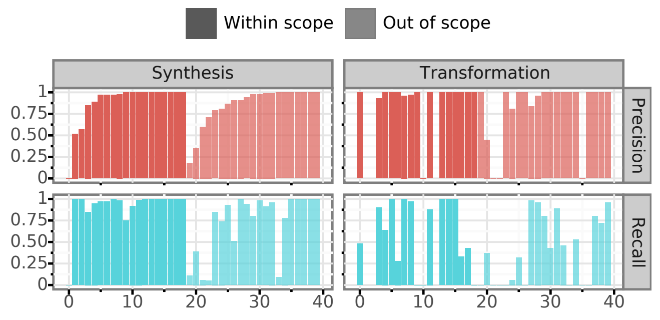

Figure 7: CFG-nodes metrics for the *challenging dataset*

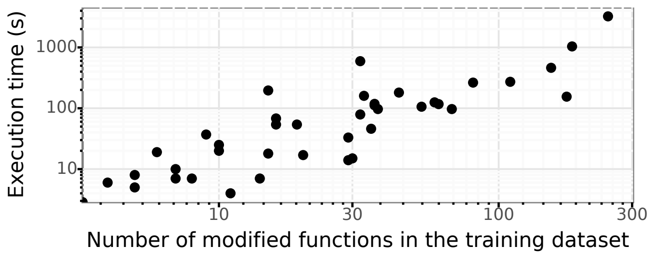

Figure 8: Execution time for the *challenging dataset*

synthesis and 81% precision and 39% recall in transformation.

These results suggest that, for the subset of examples in its scope, Spinfer can be used to infer semantic patches that can perform most of the needed transformations. Also, by carefully examining the produced semantic patches, we believe that most of the non-exhaustive semantic patches in this subset can be modified in a couple of minutes to obtain complete ones. For examples outside Spinfer scope, Spinfer is still able to perform some of the transformations without producing too many false positives.

As shown in Figure 8, Spinfer's execution time is almost linear in the number of modified functions in the training dataset. Spinfer takes less than 50 seconds to infer a semantic patch for examples with 10 or fewer modified functions. We believe that Spinfer's execution time is quite reasonable to be used as a tool to suggest a semantic patch.

4.3 Experiments on the *2018 dataset*

Figure 6 also shows the taxonomy of the changes for the *2018 dataset*. Contrary to the *challenging dataset*, the largest part of the changes in the *2018 dataset* do not require taking complex control flow into account (C_0 and C_1) and thus fit in Spinfer's scope. The kinds of control-flow constraints observed are from one statement to the next, and from one statement to some later statement that is not directly contiguous. Only a few changes require taking into account negative information (C_3), such as verification that a variable is not used in its scope before removing it, or interprocedural information (C_4). However, multiple variants (I_1) and noise (N_1) are very common.

We evaluate the results in the same way as for the *challenging dataset*. Figure 9 gives the recall and precision for the generated semantic patch. For the four C_4 cases in the taxonomy the results are evaluated against the developers' changes directly, rather than against human-written semantic patches, as Coccinelle does not support interprocedural control-flow constraints.

For the *2018 dataset*, the generated semantic patches achieve 85% precision and 83% recall in synthesis, and 87% precision and 62% recall in transformation. Looking only at the examples in the scope of Spinfer, the generated semantic patches obtained 92% precision and 88% recall in synthesis,

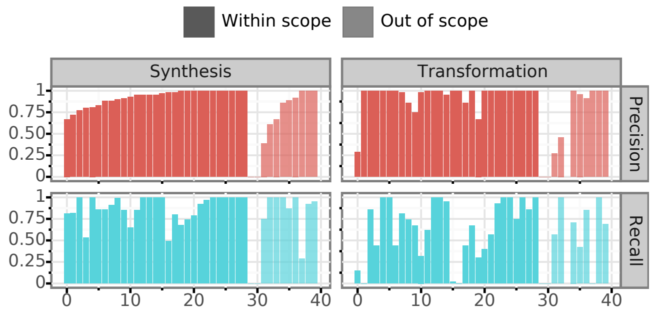

Figure 9: CFG-nodes metrics for the *2018 dataset*

and 94% precision and 65% recall in transformations, which is very close to the results for the *challenging dataset*. As most of the transformations in our randomly sampled dataset are in Spinfer scope, these results suggest that Spinfer is adapted to real kernel transformation situations, and can generate high quality semantic patches that can be used by kernel developers to perform synthesis or transformation tasks.

4.4 Analysis of a failure case

We now analyze a typical failure case that gives an intuition for why some examples have a lower recall in transformation than in synthesis.

A case where Spinfer does not generate a high quality semantic patch is for the elimination of the **cpufreq_-frequency_table_target** function. For this transformation our human expert wrote the following semantic patch:

```
@@ expression policy, old_freq; @@
- cpufreq_frequency_table_target(policy, old_freq,
-       CPUFREQ_RELATION_H)
+ cpufreq_table_find_index_h(policy, old_freq)

@@ expression policy, old_freq; @@
- cpufreq_frequency_table_target(policy, old_freq,
-       CPUFREQ_RELATION_L)
+ cpufreq_table_find_index_l(policy, old_freq)

@@ expression policy, old_freq; @@
- cpufreq_frequency_table_target(policy, old_freq,
-       CPUFREQ_RELATION_C)
+ cpufreq_table_find_index_c(policy, old_freq)
```

The semantic patch composed of three rules replaces the uses of **cpufreq_frequency_table_target** by specialized **cpufreq_table_find_index** functions. The choice of replacement function depends on the constant used as the third argument of **cpufreq_frequency_table_target**.

For this transformation we launched Spinfer on a learning set of only two files and, in under two seconds, Spinfer generated the following semantic patch composed of two rules:

```
@@ expression E0, E1, E2; @@
- E0 = cpufreq_frequency_table_target(E1, E2,
-       CPUFREQ_RELATION_H);
+ E0 = cpufreq_table_find_index_h(E1, E2);
```

```
@@ expression E0, E1; @@
- E0 = cpufreq_frequency_table_target(&E1, E1.cur,
-        CPUFREQ_RELATION_C);
+ E0 = cpufreq_table_find_index_c(&E1, E1.cur);
```

The generated semantic patch does not cover all the cases, and thus it is incomplete. It illustrates two typical reasons for the failure Spinfer: **variant bias** and **over-specialization**.

Variant bias happens when only a subset of variants is present in the learning set. In this case the learning set contains only instances with **CPUFREQ_RELATION_H** and **CPUFREQ_RELATION_C** constants. Consequently Spinfer did not see the third constant and cannot generate a rule for it. Given this semantic patch, however, a developer can easily find the missing constant and complete the semantic patch with copy-pasting and small edits.

Over-specialization is present here in two forms: all rules are assignments and the second rule contains a mandatory field **cur** for the second parameter of both functions. Both of these constraints are incorrect, but they are generated because they are present in all examples in the learning set and because Spinfer prefers precision over recall. Once again, fixing these issues can be very quick for a Linux kernel developer, once the basic outline of the semantic patch is provided.

We have illustrated two typical reasons for the failure of Spinfer, in which our tool generates an incorrect semantic patch for the testing set. These cases happen because Spinfer somewhat overfits the learning set to prevent false positives. However these kinds of failure can be easily and quickly fixed by developers, by using their knowledge of the kernel or by providing more examples to our tool.

5 Related Work

The most closely related works are Sydit [17], LASE [18], APPEVOLVE [7], REFAZER [23], MEDITOR [30], A4 [11], and GENPAT [8], that were already presented in Section 2.

A novelty of Spinfer is its focus on control-flow graphs. CBCD [14] relies on Program Dependence Graphs (PDGs) to find other instances of a known bug in a code base. CBCD is limited to only one example bug, does not produce a script for matching buggy code, and does not address how to fix bugs. By considering commonalities among multiple examples, Spinfer can learn more general rules. By providing a transformation script, Spinfer makes the result understandable to the user and even allows the user to improve the script or adapt it to other uses. Finally, Spinfer-inferred semantic patches both find code needing transformation and describe how to transform it automatically.

Spinfer requires the existence of patches that illustrate a desired change. When the developer does not know how to make a change, he can search for examples in the commit history. Patchparse [20] finds common changes in commit histories of C code. SysEdMiner [19] finds such changes for Java code. Prequel allows the developer to search for commits that match patterns of removed and added code, making it possible to find examples for specific changes [13]. The output from those tools may be used with Spinfer.

6 Conclusion

This paper proposes an approach to automatically inferring Coccinelle semantic patches from a collection of change examples written in C. Our approach considers similar code fragments and control flows among the changes to identify change patterns and construct transformation rules. Generated semantic patches are easily understandable by developers and can be used both for understanding past changes and to perform large scale transformation, with little to no modifications. We have implemented our approach as a tool named Spinfer, and evaluated it on two sets of 40 real changes from the Linux kernel. Our evaluation shown that Spinfer is capable of handling the majority of the real Linux kernel transformation situations by generating semantic patches with high recall and precision in only a few minutes.

We have also identified a taxonomy of challenges that transformation-rule inference tools for systems code must solve, based on the complexity of the control and data flow, and the number of change instances and variants. Such a taxonomy provides level ground for comparing the capabilities of different transformation-rule inference tools, and Spinfer achieves much safer and more comprehensive results than previous tools with respect to the properties defined in the taxonomy. The taxonomy can also be used to guide the future development of Spinfer and other transformation-rule inference tools. In particular, the next frontier for transformation rule inference is to effectively take into account unmodified terms into control-flow dependencies. Typically, for a large-scale change, each changed function contains many times more unmodified lines than changed lines, and the unmodified lines exhibit much greater variety. A challenge is to identify which of the unmodified lines, if any, are relevant to controlling when a change should be performed.

Acknowledgements

This research was supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003) and the ANR ITrans project. We would like to thank the anonymous reviewers and our shepherd, Baris Kasikci, for their feedback.

Availability

Spinfer source code is publicly available at https://gitlab.inria.fr/spinfer/spinfer.

References

[1] Jesper Andersen and Julia L. Lawall. Generic patch inference. In *ASE*, pages 337–346, 2008.

[2] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *ASE*, pages 382–385, 2012.

[3] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *POPL*, pages 114–126, 2009.

[4] Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*. Citeseer, 2009.

[5] Kees Cook. treewide: init_timer() -> setup_timer(), October 2017. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b9eaf1872222.

[6] Deepa Dinamani. vfs: change inode times to use struct timespec64, May 2018. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95582b008388.

[7] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, page 12, 2019.

[8] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. Inferring program transformations from singular examples via big code. In *ASE*, 2019. To appear.

[9] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *ICSE*, pages 351–360, 2011.

[10] Michael Kerrisk. Ks2012: Kernel build/boot testing, September 2012. https://lwn.net/Articles/514278/.

[11] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. A4: Automatically Assisting Android API Migrations Using Code Examples. *arXiv:1812.04894 [cs]*, December 2018. arXiv: 1812.04894.

[12] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX ATC*, pages 601–614, 2018.

[13] Julia Lawall, Derek Palinski, Lukas Gnirke, and Gilles Muller. Fast and precise retrieval of forward and back porting information for Linux device drivers. In *USENIX ATC*, pages 15–26, 2017.

[14] Jingyue Li and Michael D. Ernst. CBCD: cloned buggy code detector. In *ICSE*, pages 310–320, 2012.

[15] Edward S. Lowry and Cleburne W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[16] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With GNU Diff and Patch*. Network Theory Ltd, January 2003.

[17] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[18] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[19] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *Mining Software Repositories (MSR)*, pages 248–256, 2017.

[20] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys*, pages 59–71, 2006.

[21] Gordon D Plotkin. A note on inductive generalization.

[22] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5(1):135—151, 1970.

[23] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *ICSE*, pages 404–415, 2017.

[24] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. *arXiv*, 2018.

[25] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, November 1987.

[26] Claude Sammut and Geoffrey I. Webb, editors. *TF–IDF (Encyclopedia of Machine Learning)*, pages 986–987. Springer US, Boston, MA, 2010.

[27] Wolfram Sang. tree-wide: simplify getting .drvdata, April 2018. https://lkml.org/lkml/2018/4/19/547.

[28] Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/v5.4/process/submitting-patches.html.

[29] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automated deprecated-api usage update for android apps: How far are we? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 602–611. IEEE, 2020.

[30] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of API migration edits. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 335–346, Piscataway, NJ, USA, 2019. IEEE Press.

[31] Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 515–524. ACM, 2002.