Noname manuscript No.
(will be inserted by the editor)

Deep Learning-Based Automatic Semantic Patch Generation for the Linux Kernel: How Far Are We?

Imam Nur Bani Yusuf[1] · Xiping Huang[1] · Julia Lawall[2] · Lingxiao Jiang[1] · David Lo[1] · Nghi D. Q. Bui[3] · Chunmiao Li[4] · Haitao Wu[5] · Yijun Yu[6]

Received: date / Accepted: date

Abstract The Linux kernel is a massive software system with tens of millions of lines of code and many variants. Making consistent code changes across these versions poses significant challenges. Coccinelle assists developers in transforming C code through program transformation rules written in Semantic Patch Language (SmPL), but many still struggle with its syntax and defining appropriate abstractions. Previous work introduced Spinfer, a tool that automatically generates semantic patches by mining patterns from sample changes. While useful, Spinfer has limitations: (1) it relies on specific heuristics which restricts the types of patches it can produce, and (2) it requires a sufficient number of sample changes to produce high-quality patches. Deep learning has shown impressive results across many software engineering tasks, but its potential for semantic patch generation remains unexplored. This research fills this gap by evaluating the effectiveness of deep learning in generating semantic patches, seeking alternatives to Spinfer. We have conducted experiments comparing Spinfer with deep learning models (GPT-4, Claude, and DeepSeek) on real-world kernel changes. Our results reveal that each approach performs differently in handling code abstraction and control flow dependencies across different change patterns. Through systematic error analysis, we identify and categorize distinct failure patterns. Deep learning models generate valid semantic patches with minimal examples but struggle with syntax correctness, while Spinfer ensures syntactic validity but creates overly specific patches with unnecessary constraints. Further, both approaches perform poorly on diverse change examples with multiple variants. Based on our findings, we propose several directions for improving semantic patch generation, including developing hybrid approaches combining the strengths of both methods, enhancing handling

[1]School of Computing and Information Systems, Singapore Management University
imamy.2020@phdcs.smu.edu.sg, {xphuang, lxjiang, davidlo}@smu.edu.sg
[2]Inria, Paris, France. julia.lawall@inria.fr
[3]Independent Researcher. bdqnghi@gmail.com
[4]Beijing Academy of Blockchain and Edge Computing, China. chunmiaoli1993@gmail.com
[5]Independent Researcher. wain303009@hotmail.com
[6]The Open University, UK. y.yu@open.ac.uk

of variant diversity, and creating improved mechanisms for determining optimal abstraction levels in semantic patches.

Keywords semantic patches, coccinelle, linux kernel, code transformation, deep learning, language models

1 Introduction

The Linux kernel is present in many computing applications, such as servers, embedded systems, smartphones, and supercomputers. Developers often revise the codebase to cater to new requirements, improve security, or enhance maintainability. After more than 30 years of development, the Linux kernel now has tens of millions of lines of code with many versions and variants. A single change to the Linux kernel, e.g., a function interface change, may trigger the need for repetitive modifications scattered across the codebase [12, 13, 18, 21].

Figure 1 shows a diff of real-world code changes due to the uses of `setup_timer` in the Linux kernel to simplify the initialization of Linux kernel timers. These changes highlight several key patterns. First, the modification can have equivalent semantics expressed through different syntax. Changes 1 and 2 demonstrate this, where both diffs use `setup_timer` but with different arguments. Moreover, there are different unchanged lines between the modified lines (lines 4–6 in Change 1 versus line 12 in Change 2). Second, the modification can have equivalent semantics but different control flow orderings; e.g., Changes 1 and 3 both achieve the same timer setup but their removed lines involving `.data` and `.function` are in different sequences. The migration from `init_timer` to `setup_timer` started with 73 changes in 2008, followed by minimal activity until 2014. Activity then resumed with 43, 93, and 37 changes in 2014-2016 respectively, before concluding with the removal of 267 calls to `init_timer` in 2017 [26]. These changes demonstrate the need to automate repetitive code changes, particularly when handling variants of functionally similar code changes across large codebases.

One way to propagate similar code changes across large codebases is to utilize Coccinelle [12, 13, 18, 21]. Coccinelle allows a developer to describe desired changes as a *semantic patch* in its specific *Semantic Patch Language* (SmPL), and automatically apply the semantic patch to an entire codebase. A semantic patch consists of a set of transformation rules, expressed in a patch-like format, and each of which can declare and use metavariables to abstract over some terms. Given a well-defined semantic patch, Coccinelle identifies and updates the locations in the codebase that match the specified rules. It has been widely adopted by developers, with over 10,000 Linux kernel commits made using Coccinelle [12] since 2008. However, writing semantic patches can be difficult for developers who are unfamiliar with the unique SmPL syntax and the complexity of the codebase, which provokes the need for automatic semantic patch generation from concrete code change examples.

Serrano et al. [26] introduced Spinfer as a tool to infer semantic patches from a collection of code samples. Spinfer employs clustering and pattern mining techniques to identify heuristic patterns in similar code changes and take into account variations in control flows. Nevertheless, it requires a significant number of change examples to infer correct semantic patches and may produce over-fitting semantic patches if irrelevant code fragments are present in the change examples.

```
// Change 1 shown in a diff format:
- init_timer(&dev→catas_err.timer);
+ setup_timer(&dev→catas_err.timer, poll_catas, (unsigned long)dev);
  dev→catas_err.map = NULL;
  addr = pci_resource_start(dev→pdev, 0) + ((pci_resource_len(dev→pdev, 0) - 1) & dev→
       catas_err.addr);
  ...
- dev→catas_err.timer.data = (unsigned long) dev;
- dev→catas_err.timer.function = poll_catas;

// Change 2 shown in a diff format:
- init_timer(&lanai→timer);
+ setup_timer(&lanai→timer, lanai_timed_poll, (unsigned long)lanai);
  lanai→timer.expires = jiffies + LANAI_POLL_PERIOD;
- lanai→timer.data = (unsigned long) lanai;
- lanai→timer.function = lanai_timed_poll;

// Change 3 shown in a diff format:
- init_timer(&psw→debounce);
- psw→debounce.function = switch_timer;
- psw→debounce.data = (unsigned long)psw;
+ setup_timer(&psw→debounce, switch_timer, (unsigned long)psw);
```

Fig. 1 Three sample changes that simplify **`init_timer`** to **`setup_timer`**. They involve different function arguments; Diff 3 has different ordering of **`.function`** and **`.data`** data fields.

This paper investigates the feasibility of using deep learning models to automatically generate semantic patches, especially for scenarios where many locations in large codebases need diverse variants of code changes, but only a limited number of sample changes are available. By studying the effectiveness of deep learning models (GPT-4, Claude, and DeepSeek) on semantic patch generation in comparison to Spinfer, we provide insights into the strengths and weaknesses of different approaches in dealing with the challenges associated with diverse syntactic and semantic patterns in code changes, and propose possible future improvements to address the challenges.

The main contributions of this paper are as follows:

- We are the first to explore the use of deep learning models for semantic patch generation and systematically evaluate their effectiveness against Spinfer on real-world Linux kernel code changes of diverse complexities.
- Through extensive manual error analysis, we identify distinct failure patterns of Spinfer and deep learning models to shed light on their strengths and weaknesses, and distill several key lessons for future research directions, including developing hybrid approaches combining strengths of different methods, enhancing handling of variant diversity, improving mechanisms for determining optimal abstraction levels of semantic patches, and practical integration of semantic patch generation tools.

The rest of the paper is organized as follows. Section 2 elaborates on semantic patches in detail and discusses the challenges in generating them. Section 3 explains our study design, datasets, and experimental settings. Section 4 presents our empirical results, followed by our error analysis in Section 5. Section 6 outlines lessons learned and future directions. Section 7 discusses threats to validity, while Section 8 discusses related work. Finally, Section 9 concludes the paper.

Data availability. The script, artifacts, and dataset used in our study are available at `https://github.com/imamnurby/SMPL-How-Far-Are-We`.

2 Motivating Examples & Research Questions

We study the scenarios when a developer needs to implement changes across a large codebase, e.g., migrating all the uses of init_timer to setup_timer (Figure 1) or fixing all instances of a bug in the codebase. The developer may start with manual modifications in several locations, resulting in a set of semantically equivalent change examples. However, given the massive scale of the codebase (e.g., millions of lines of code in the Linux kernel), the developer cannot be certain that these manual changes are complete. Manually identifying and updating every affected location is both time-consuming and error-prone. Instead, the developer can describe the needed code transformation using a semantic patch and use Coccinelle to automatically propagate these changes across the codebase. Using semantic patches can not only reduce the effort required to find all affected locations but also ensure consistency across all changes. However, writing semantic patches requires expertise in both the codebase and Coccinelle's SmPL language, and can be challenging for developers to write manually.

Our study investigates to what extent existing tools can generate semantic patches automatically from a few code change examples so as to save developers' manual efforts in writing semantic patches and propagating all changes. Currently, Spinfer represents the state-of-the-art in automated semantic patch generation. Spinfer leverages pattern mining techniques to identify common transformation fragments and control-flow constraints between matched code fragments, then assemble them into semantic patch rules. Alongside these rule-mining-based approaches, recent advances in deep learning also show significant potential for automatically generating semantic patches by learning transformation patterns from change examples.

Figure 2 shows a semantic patch for the change examples in Figure 1. A semantic patch consists of a set of transformation rules (lines 5–9 and lines 14–18) and their metavariable declarations (line 3 and line 12). We briefly explain the semantic patch rules here. The three metavariables, t, d, and f, are declared to match expressions in the code: t matches the function argument of init_timer and is used as the first argument of setup_timer; d matches the expression assigned to the .data field and is used as the third argument of setup_timer; f matches the expression assigned to the function field and is used as the second argument of setup_timer. The dot notation (... , lines 7 and 16) supported by Coccinelle abstracts away the code along the control-flow path between the init_timer invocation and the field assignment (lines 5–8 and lines 14–17). The rule for Change 3 is separated from the rule for Changes 1 and 2 in Figure 1, as the removal order of the function and data fields is different and SmPL does not directly support the description of alternative code orders.

```
1  // Alternative 1
2  // Semantic Patch for Changes 1 and 2
3  @@
4  expression t,d,f;
5  @@
6  - init_timer(&t);
7  + setup_timer(&t,f,d);
8  ...
9  - t.data = d;
10 - t.function = f;
11
12 //Semantic Patch for Change 3
13 @@
14 expression t,d,f;
15 @@
16 - init_timer(&t);
17 + setup_timer(&t,f,d);
18 ...
19 - t.function = f;
20 - t.data = d;
```

```
1  // Alternative 2
2  // Semantic Patch for Changes 1 and 2
3  @@
4  expression t,d,f;
5  identifier e;
6  @@
7  - init_timer(&e→t);
8  + setup_timer(&e→t,f,d);
9  ...
10 - e→t.data = d;
11 - e→t.function = f;
12
13 //Semantic Patch for Change 3
14 @@
15 expression t,d,f;
16 identifier e;
17 @@
18 - init_timer(&e→t);
19 + setup_timer(&e→t,f,d);
20 ...
21 - e→t.function = f;
22 - e→t.data = d;
```

Fig. 2 Two possible semantic patches with different abstractions for the code changes in Figure 1.

2.1 Challenges in Automatic Semantic Patches Generation

Generating semantic patches automatically from code change examples presents two main challenges: 1) determining the correct level of abstraction from change examples and inferring control-flow dependencies, and 2) generating different transformation rules for multiple change variants. We now detailing these challenges and illustrate them through our running example.

Determining the correct level of abstraction from change examples and inferring control-flow dependencies. For example, in Figure 1, Changes 1 and 2 illustrate how a semantic patch generation method must identify similar changes between input examples, such as the substitution of **`init_timer`** with **`setup_timer`**, along with the removal of the **`data`** and **`function`** field assignments. After detecting the common changes, the method must infer the appropriate abstractions for the common changes using metavariables. As shown in Figure 2, different levels of abstraction can be inferred based on the code change examples in Figure 1. Alternative 1 is more abstract, where the argument of **`init_timer`** can be any expression. In contrast, the argument of **`init_timer`** in Alternative 2 is more specific, as it must have the form of a field access **`e->t`**. The method must also determine which code fragments should be abstracted away or included as the context for the transformation. In Changes 1 and 2 of Figure 1, both share the same sequence of removals and additions but contain unrelated code between lines 3–7 in Change 1 and lines 11–13 in Change 2. The method must decide when these other lines contain information that has an impact on the transformation and when to abstract away these intervening lines using dot notation (`...`) to represent other code between the changed elements.

Generating different transformation rules for multiple change variants. A variant is a distinct implementation of a code change that differs in control

flow, ordering, or structure, but has the same semantics. In our example above, two variants exist: one in Changes 1 and 2, and another in Change 3, and their key difference is the order of the field removals—Changes 1 and 2 remove the `data` field first and then the `function` field, while Change 3 reverses this order. This difference requires the semantic patch generation method to recognize these variations and generate specific rules for each variant, as shown in Figure 2.

2.2 Spinfer & Its Limitations

The prior approach, Spinfer, infers semantic patches based on traditional clustering and pattern mining techniques. Its performance is still limited in addressing the challenges mentioned above and it is unable to handle complicated cases. Spinfer automates the inference of Coccinelle transformation rules from a set of example code changes. It follows a three-step process: (1) identifying abstract fragments by clustering structurally similar subterms, (2) assembling these fragments into rule-graphs based on control-flow dependencies, and (3) refining the rules through splitting and ordering to maximize precision and recall. The final output is a set of transformation rules that capture common patterns in the provided examples.

Despite its effectiveness, Spinfer has several limitations. First, Spinfer leverages agglomerative hierarchical clustering and represents code fragments using FF-ITF (Function Frequency-Inverse Term Frequency), which is similar to TF-IDF but uses function matching instead. Once two code fragments are placed in different clusters, they can never be rejoined even if they share common elements at a finer granularity. Second, it requires a sufficient number of change examples to produce semantic patches that can generalize beyond the provided input examples because the default is to be as specific as is possible to match all the input examples.

2.3 Our Research Questions

While deep learning approaches have demonstrated success in various software coding tasks, their effectiveness for semantic patch generation remains unexplored compared to traditional approaches like Spinfer. In this paper, we aim to investigate and provide answers to the main research question: *How well can deep learning models, such as GPT-4, Claude, and DeepSeek, perform for the task of semantic patch generation in comparison to traditional approaches?* More specially, we want to investigate:

RQ1 How accurately do deep learning models and Spinfer capture transformation patterns from change examples used for patch inference?
RQ2 How well do the semantic patches generated by deep learning models and Spinfer generalize to unseen code?
RQ3 How do variations in the input change examples affect the performance of semantic patches generated by deep learning models and Spinfer on unseen code?
RQ4 What are the characteristic error patterns in semantic patches generated by deep learning models and Spinfer?
RQ5 What lessons can be learned from our study, and what are the promising future research directions in this field?

3 Study Design

Our study consists of two main stages: patch inference and patch application. In the patch inference stage, we aim to derive semantic patches from code transformation examples. In the patch application stage, we evaluate these inferred patches by applying them to unseen input code.

Let $E = \{(C_b^1, C_a^1), (C_b^2, C_a^2), ..., (C_b^k, C_a^k)\}$ be a set of change example pairs during the inference stage, where C_b^i represents the code before transformation, C_a^i represents the code after transformation, and k is the number of code example pairs in E. The goal is to generate a semantic patch P_o that captures the transformation pattern in the set of code transformation pairs E. The semantic patch inference process differs between the deep learning models and Spinfer due to their distinct working mechanisms. For the deep learning models, each transformation pair (C_b^i, C_a^i) is first converted into a unified diff representation, as this format closely aligns with the transformation rules in semantic patches. These diffs are then inserted into a template prompt to produce a prompt that serves as the model input. Given this input prompt, the model auto-regressively produces the raw output that contains a reasoning path and the final semantic patch. Then, the raw output is processed to extract the final semantic patch. In contrast, Spinfer directly processes the set of transformation pairs E, performing clustering on the examples and constructing the semantic patch through control flow analysis.

In the patch application stage, we evaluate the generated semantic patch P_o. Given an input C code C_{in}, Coccinelle determines whether P_o is applicable. If applicable, Coccinelle transforms C_{in} and produces the output code C_{out}. Otherwise, no transformation occurs, indicating that P_o is not applicable to C_{in}.

The following sections describe the dataset used for both patch inference and application, outline our prompt design for the deep learning model, and present the evaluation metrics used to assess both the inferred semantic patches and the transformed code.

3.1 Dataset Overview

Our dataset consists of real-world C source file changes from the Linux kernel codebase, obtained from a previous study [26]. Each instance contains a pair of C code snippets representing the code before and after a change. These changes are organized into sets of change examples. Figure 3 provides an overview of a set of change examples. Each collection can have multiple variants. A variant is a distinct implementation of a code change that differs in control flow, ordering, or structure, while maintaining the same change semantics (see Figure 1). Each variant is represented by a set of changes (C before and after) that occur across various locations in the codebase.

Due to budget constraints for deep learning API calls, we randomly sampled 14 sets of change examples from the previous study [26]. While the original study was conducted at the file level, we focus on function-level changes for two reasons: (1) files typically contain multiple functions but changes often affect only specific functions, and (2) function-level analysis reduces input length, helping manage the deep learning API costs within our budget constraints.

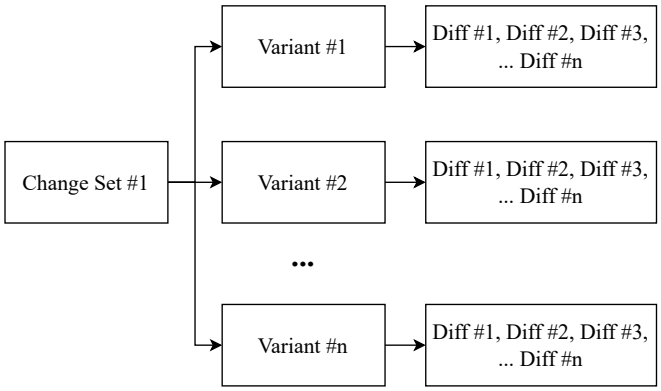

Fig. 3 The overview of the dataset structure.

Table 1 Dataset statistics. FN=Function, Avg=Average, LoC=Lines of Code.

Change Set	# Fn	# Variant	Avg. LOC Diff	Avg. Tokens	Max Tokens
EXP0-7	21	2	62.62	463.24	2710
dasd_smalloc	23	2	72.70	528.48	1457
dma_pool_alloc-52	11	1	99.82	727.73	1896
early_memunmap	11	1	23.45	171.36	252
free_bootmem-77	13	1	21.08	151.46	491
kees_timer1	26	3	50.65	395.50	1305
perf_evlist__mmap-69	14	1	93.36	602.79	1311
random_ether_addr-84	15	1	59.67	428.53	958
snd_soc	26	2	22.92	180.42	375
sock_poll_wait-84	13	1	24.23	199.62	324
tcaction	12	1	50.42	314.00	574
tcf_block_get-61	20	2	48.75	415.30	1188
ttm_bo_init-60	9	1	32.11	299.33	719
uartlite	9	1	12.22	87.78	218
Total	223	27	49.44	366.53	2710

Preprocessing. We extract differences between code before and after transformation using the difflib library.[1] We use the unified diff format as it resembles the semantic patches syntax (see Figure 1). We extract each modified function, omitting unchanged ones. The resulting function-level diffs serve as input for deep learning models. For Spinfer, we use the function-level code before and after transformation as input. Table 1 presents the statistics for each change set.

Dataset Usage. We divide the change examples in each change variant within each change set into two splits: inference and test. The inference split consists of k change examples sampled from a total of n change examples for each variant in each change set, forming the set E. This set E is used to infer the semantic patch for each change set. We set $k \in \{1, 2, 3\}$, reflecting realistic scenarios where

[1] `https://docs.python.org/3/library/difflib.html`

limited change examples are available. The remaining $n-k$ change examples form the test split, used to evaluate how well the generated semantic patches generalize to unseen code patterns. To ensure fair comparison across different values of k, we maintain consistent test splits for each change set across different k values, allowing us to assess how increasing the number of inference examples affects generalization performance. In the end, we have 170 change examples in the test splits. The remaining 60 change examples are used for patch inference.

3.2 Evaluation Metrics

We evaluate both the generated semantic patches and their resulting code transformations. For each semantic patch P, we assess its validity and transformation accuracy using precision and recall. These measurements are then aggregated across all patches to evaluate the overall performance.

Patch Validity determines whether a semantic patch can be parsed by the Coccinelle parser. For each patch P, the validity score v_i is binary, as shown in Equation 1:

$$\text{Validity} = \begin{cases} 1 & \text{if } P \text{ can be parsed by Coccinelle} \\ 0 & \text{if } P \text{ is invalid or absent} \end{cases} \tag{1}$$

Precision and Recall measure alignment between transformed code and ground truth. Precision quantifies how many modifications made by the generated patch are correct. A high precision indicates the semantic patch introduces few incorrect changes. Recall measures how many expected modifications are captured by the generated semantic patch. A high recall means the semantic patch correctly performs most or all necessary modifications. For each semantic patch P applied to code C_{in}, we compute the average precision and average recall using Equation 2 and Equation 3, respectively. Here, $\Delta\text{Lines}_{\text{result}}$ represents lines modified by the generated patch, while $\Delta\text{Lines}_{\text{truth}}$ represents lines modified in the ground truth. Both metrics range from 0 to 1, with higher values indicating better performance.

$$\text{Precision} = \frac{|\Delta\text{Lines}_{\text{result}} \cap \Delta\text{Lines}_{\text{truth}}|}{|\Delta\text{Lines}_{\text{result}}|} \tag{2}$$

$$\text{Recall} = \frac{|\Delta\text{Lines}_{\text{result}} \cap \Delta\text{Lines}_{\text{truth}}|}{|\Delta\text{Lines}_{\text{truth}}|} \tag{3}$$

Overall Performance. Overall performance is determined by aggregating individual metrics across all evaluated change examples. Let T be the total number of change examples. For each metric, we compute the average validity, precision, and recall using Equation 4, Equation 5, and Equation 6. We report performance at two granularities: change set and global. In the change set-level evaluation, we compute metrics separately for each semantic patch within its respective change set. In the global-level evaluation, we compute metrics across all change examples in all change sets, providing a unified performance measure. Formally, for change set-level evaluation, T represents the number of change examples within

each change set, while for global-level evaluation, T represents the total number of change examples across all change sets. We use the arithmetic mean for aggregation instead of the geometric mean since some change sets may yield zero results, which would make the geometric mean zero. Additionally, the relatively modest variation in the change example counts across change sets (only 2-3x difference) ensures the arithmetic mean is not dramatically skewed by any single change set.

$$\text{Validity}_{\text{avg}} = \frac{1}{\text{T}} \sum_{i=1}^{T} \text{Validity}_{\text{i}} \tag{4}$$

$$\text{Precision}_{\text{avg}} = \frac{1}{\text{T}} \sum_{i=1}^{T} \text{Precision}_{\text{i}} \tag{5}$$

$$\text{Recall}_{\text{avg}} = \frac{1}{\text{T}} \sum_{i=1}^{T} \text{Recall}_{\text{i}} \tag{6}$$

Preprocessing Before Metric Computation. The transformation output from Coccinelle may have different formatting compared to the ground truth, particularly in line breaks. These formatting inconsistencies can affect diff generation and consequently our precision and recall calculations. To address this issue, we use ClangFormat[2] to standardize both the generated output and ground truth C files before computing metrics.

3.3 Prompt Engineering for Deep Learning Models

Prompt engineering guides language models to generate desired outputs through carefully crafted input prompts. This technique is valuable when direct model tuning is impossible, such as with closed-source models like GPT-4 and Claude or when facing memory constraints. Our prompt comprises four main sections: 1) instruction, 2) input diffs, 3) output format, and 4) background knowledge.

The instruction section contains the directive to generate semantic patches for Coccinelle using the SmPL language. The input diffs section provides example transformations in unified diff format showing code before and after the transformation. Using this information, the model must generalize the change pattern to generate the semantic patch. The output format section specifies how the model should structure its response. We require the model to first analyze the input diff and provide its reasoning. This analytical step before generating the final output has been shown to enhance model performance [31].

To ensure the model understands domain-specific terminology, we include a background knowledge section providing more context on Coccinelle and semantic patches. This context includes: Coccinelle's motivation and goals that are obtained from from the official documentation[3], and semantic patch information from the

[2] `https://clang.llvm.org/docs/ClangFormat.html`

[3] `https://coccinelle.gitlabpages.inria.fr/website/ce.html`

grammar documentation.[4] We also include some change examples and the corresponding semantic patches from the official documentation in our prompt. Obtaining information from the official documentation makes sense as it is written by domain experts who developed the tool and is specifically intended for end-users, ensuring accuracy and proper context. The semantic patch information covers patch structure, explanation of metavariables and transformation rules, and valid metavariable types. Figure 4 shows the prompt used in our study. The complete prompt can be accessed in our replication package.[5]

3.4 Implementation Details

For our experiments, we use GPT-4 version gpt-4-2024-11-20, Claude-3-7-sonnet-20250219, and DeepSeek-V3. We choose these models because they perform well in various tasks and were released in the past 6 months. We maintain default temperature and top-p parameters across all models. The maximum token generation limit is set to 4,096. This limit is set based on our preliminary experiment, where each model generates around 1000 output tokens on average. We provide an additional 3000 tokens as buffer in case of handling complex diffs. To address potential API failures, we configure a maximum of two retries per call. For each successful call, we generate three responses and select the optimal semantic patch by measuring precision and recall when applied to the initial C files from input diffs in the prompt. All model responses are recorded and included in our replication package to ensure transparency and reproducibility. For Spinfer, we utilize its official implementation available at this URL.[6]

4 Empirical Evaluation Results

4.1 RQ1: How accurately do deep learning models and Spinfer capture transformation patterns from change examples used for patch inference?

We evaluate the accuracy of each approach in generating semantic patches that correctly reproduce transformations from the input change examples. Our evaluation process consists of three steps: (1) assessing if the generated semantic patches are syntactically valid, (2) applying valid patches to initial code C_{in} to produce transformed code C_{out}, and (3) comparing C_{out} with ground truth transformed code C_{truth} using precision and recall metrics. We evaluate the performance of each approach with varying numbers of change examples (k), ranging from 1 to 3. First, we examine the global-level patch validity, precision, and recall for each approach in Section 4.1.1. Then, we present and discuss the performance per change set in Section 4.1.2.

[4] https://coccinelle.gitlabpages.inria.fr/website/docs/main_grammar.pdf

[5] https://github.com/imamnurby/SMPL-How-Far-Are-We/blob/main/emse_submission_fix/prompt.txt

[6] https://gitlab.inria.fr/spinfer/spinfer

Task
Write a semantic patch for Coccinelle using the SmPL (Semantic Patch Language) based on all diffs specified under **# Input Diffs**. The background knowledge needed to generate semantic patches is specified under **# Background Knowledge**.

Input Diffs
{insert diff(s) here. The number of provided diffs can vary depending on avaialble change examples.}

Output Format
Output the analysis before the final answer.

Background Knowledge
Motivation of Coccinele: Collateral Evolutions
The Linux operating system (OS) is evolving rapidly to improve performance and to provide new features. This evolution, however, ...

Coccinelle
Coccinelle is a program transformation engine developed to address the problem of collateral evolution in drivers. The key idea of Coccinelle is to ...

Semantic Patches
... Semantic patches, and the associated transformation engine spatch, abstract away:
- Differences in spacing, indentation, and comments
- Choice of names given to variables (use of metavariables)
...

Semantic Patch Examples
{Few examples that are taken from the official documentation at https://coccinelle.gitlabpages.inria.fr/website/sp.html}

Semantic Patch Structure
A semantic patch consists of a metavariable definition and a rule definition.
@@
// Metavariable definition
[the metavariable definition is here]
@@
// Rule definition
[the rule is here]
{Explanation of the metavariable and rule}

Valid Metavariable Types
- identifier: matches the name of a structure field, macro, function, or variable. It can also represent a variable in an expression. ...

Fig. 4 The simplified prompt that we use in our study.

4.1.1 Global Performance

Table 2 compares the patch validity scores of GPT-4, DeepSeek, Claude, and Spinfer across different numbers of change examples (k). Both Claude and Spinfer demonstrate improved patch validity when provided with more change examples. respectively. In contrast, GPT-4 and DeepSeek show inconsistent patterns, with their patch validity scores fluctuating as the number of examples increases.

Table 2 Patch validity comparison across GPT-4, Claude, DeepSeek, and Spinfer models on the inference dataset. The generated semantic patches are classified into three categories: Valid (successfully parsed by Coccinelle), Invalid (cannot be parsed by Coccinelle), and No Result (no semantic patches generated).

#Examples (k)	Approach	Valid	Invalid	No Result
1	GPT-4	0.71	0.29	0.00
	Claude	0.64	0.36	0.00
	DeepSeek	0.71	0.29	0.00
	Spinfer	0.79	0.00	0.21
2	GPT-4	0.57	0.43	0.00
	Claude	0.79	0.21	0.00
	DeepSeek	0.64	0.36	0.00
	Spinfer	0.86	0.00	0.14
3	GPT-4	0.64	0.36	0.00
	Claude	0.79	0.21	0.00
	DeepSeek	0.57	0.43	0.00
	Spinfer	0.86	0.00	0.14

GPT-4's performance in generating valid semantic patches shows varying patch validity across different numbers of change examples (k). With k=1, GPT-4 successfully produces valid semantic patches for 10 out of 14 change sets (71%), which slightly decreases to 9 out of 14 change sets (64%) with k=2, then slightly increases again to 10 out of 14 (71%) with k=3. Similarly, DeepSeek demonstrates varying patch validity across different numbers of change examples (k). With k=1, DeepSeek achieves its highest performance, successfully producing valid semantic patches for 11 out of 14 change sets (71%). This success rate decreases to 9 out of 14 change sets (64%) with k=2, and further drops to 8 out of 14 change sets (57%) with k=3.

The patch validity for Claude and Spinfer increases with more change examples. With k=1, Claude successfully produces valid semantic patches for 9 out of 14 change sets (64%). This performance improves with k=2 and k=3, where Claude achieves success in 11 out of 14 change sets (79%) in both cases. Spinfer also demonstrates strong performance in patch validity across different numbers of change examples. With k=1, Spinfer successfully produces valid patches for 11 out of 14 change sets (79%). The performance slightly improves with k=2 and k=3, where Spinfer achieves success in 12 out of 14 change sets (86%) for both cases. Notably, Spinfer does not yield perfect patch validity because it does not produce any semantic patches for change sets **`uartlite`**, **`snd_soc`**, and **`tcf_block_get-61`** at k=1 and **`snd_soc`**, and **`tcf_block_get-61`** at k={2,3}.

Table 3 presents the overall precision and recall for GPT-4, DeepSeek, Claude, and Spinfer across different numbers of change examples (k). We measure the precision and recall in two settings: all and valid-only. In the all setting, we incorporate all change sets in our calculations, including those with invalid or no semantic patches. We assign the precision and recall values to 0.00 for these change sets. In the valid-only setting, we only consider change sets with valid semantic patches when computing the metrics, which provides insight into the quality of successfully generated patches independent of failure cases. This dual reporting approach allows us to evaluate both the models' overall effectiveness across all change sets

and their performance specifically on cases where they successfully generate valid semantic patches.

Table 3 Precision and recall comparison of GPT-4, Claude, DeepSeek, and Spinfer on the inference split, measuring how well the generated semantic patches can reproduce transformations from the change examples. **All**: Change sets with invalid semantic patches or no semantic patches are included in calculations. **Valid-only**: Change sets with invalid semantic patches or no semantic patches are excluded from calculations.

#Examples (k)	Approach	Precision		Recall	
		All	Valid-only	All	Valid-only
1	GPT-4	0.42	0.61	0.42	0.61
	Claude	0.50	0.83	0.50	0.83
	DeepSeek	0.52	0.88	0.52	0.88
	Spinfer	0.72	0.97	0.72	0.97
2	GPT-4	0.37	0.67	0.38	0.68
	Claude	0.54	0.83	0.55	0.85
	DeepSeek	0.49	0.89	0.50	0.91
	Spinfer	0.74	0.93	0.75	0.94
3	GPT-4	0.35	0.58	0.35	0.58
	Claude	0.67	0.89	0.67	0.90
	DeepSeek	0.39	0.88	0.40	0.89
	Spinfer	0.76	0.95	0.77	0.96

GPT-4 and DeepSeek show declining performance with additional examples. When all change sets are included, GPT-4 achieves precision and recall of 0.42 at k=1, which decrease to 0.37 for precision and 0.38 for recall with k=2, and further drop to 0.35 for both metrics with k=3. When excluding change sets with invalid semantic patches, GPT-4's performance is substantially higher, with precision and recall of 0.61 at k=1, increasing to 0.67/0.68 at k=2, before dropping to 0.58 for both metrics at k=3. DeepSeek similarly exhibits decreasing performance with additional examples. When all change sets are included, DeepSeek achieves the highest scores of 0.52 for both precision and recall at k=1. However, the metrics decline to 0.49/0.50 at k=2, and continue to decrease to 0.39/0.40 at k=3. When excluding change sets with invalid semantic patches, DeepSeek demonstrates remarkably high performance, achieving precision and recall of 0.88 at k=1, improving slightly to 0.89/0.91 at k=2, and maintaining 0.88/0.89 at k=3. This decline in performance metrics for both GPT-4 and DeepSeek in the "Included" column aligns with its success rate in generating valid semantic patches. Furthermore, the substantial difference between the "Included" and "Excluded" columns reveals that the decrease in precision and recall for both GPT-4 and DeepSeek is mainly due to the invalid semantic patches.

Both Claude and Spinfer demonstrate improved performance as the number of change examples increases. For Claude, the precision and recall are 0.50 at k=1 when all change sets are included. These metrics improve to 0.54/0.55 at k=2, and reach 0.67 for both metrics at k=3. When only including change sets with valid semantic patches, Claude achieves much higher scores with precision and recall of 0.83 at k=1, maintaining similar performance at k=2, and further improving to 0.89/0.90 at k=3. Spinfer exhibits the strongest overall performance among all

approaches. The precision and recall improve as the number of change examples increases when all change sets are included. Starting with 0.72 for both metrics at k=1 when the change sets with no results are included, the scores increase to 0.74/0.75 at k=2, and further improve to 0.76/0.77 at k=3. When change sets with no semantic patches are excluded, the performance fluctuate slightly around 0.93-0.97 across different number of change examples (k).

Our experimental results demonstrate significant variance in how different approaches handle semantic patch generation as the number of change examples increases. Spinfer consistently outperforms all LLM-based approaches, maintaining high precision and recall regardless of how many examples are provided. Among the LLM-based approaches, Claude shows the most promising results with a positive correlation between performance and the number of examples, while GPT-4 and DeepSeek surprisingly demonstrate declining performance with additional examples. These findings suggest that simply providing more examples does not universally improve performance for all deep learning models in the semantic patch generation task. The ability to generate valid semantic patches is a critical factor that will affect the overall effectiveness of the deep learning models in this task.

4.1.2 Change Set Performance

Figures 5, 6, and 7 show how the performance of GPT-4, Claude, DeepSeek, and Spinfer varies across different change sets in the test split with different numbers of change examples (k).

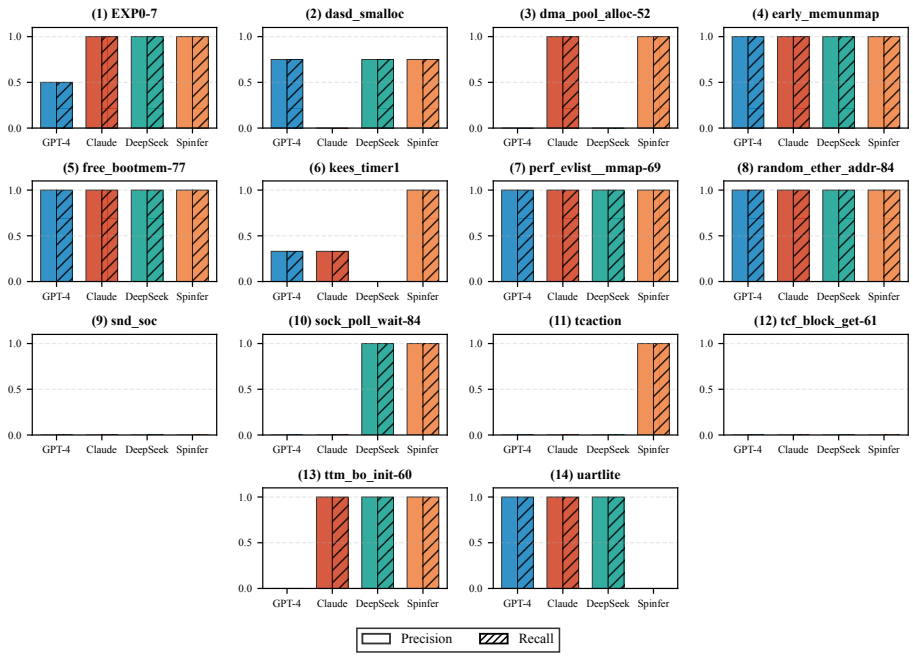

Fig. 5 Comparison of deep learning models and Spinfer across 14 test directories when using one input change example (k=1) to infer the semantic patches.

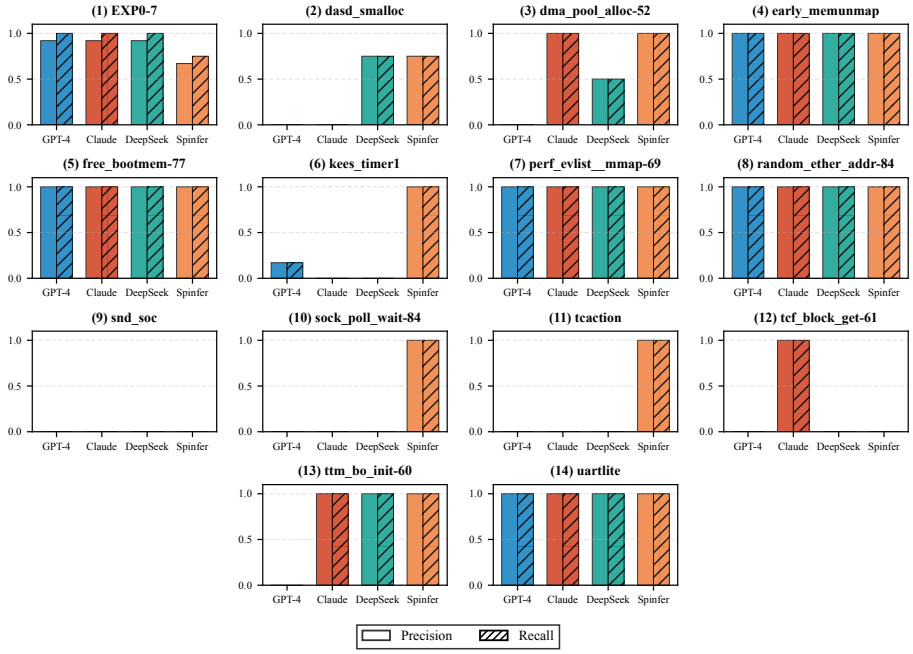

Fig. 6 Comparison of deep learning models and Spinfer across 14 test directories when using two input change examples (k=2) to infer the semantic patches.

GPT-4 consistently produces valid semantic patches that achieves perfect precision and recall (1.00) in all number of change examples (k) for: **`early_memunmap`**, **`free_bootmem-77`**, **`perf_evlist__mmap-69`**, **`random_ether_addr-84`**, and **`uartlite`**. For **`early_memunmap`**, **`free_bootmem-77`**, and **`random_ether_addr-84`**, these set of change examples contain changes involving function renaming. On the other hand, the set **`perf_evlist__mmap-69`** involves removing one function argument. Figure 8 shows the example of the semantic patch generated by GPT-4 for **`early_memunmap`**. Here, the changes are straightforward transformations, but still involve code fragments with dependencies between the removed and added lines (**`E1`** and **`E2`**).

The set **`uartlite`** also involves code fragments with dependencies between the removed and added lines. Figure 9 shows an example of the semantic patch for **`uartlite`**. Here, the transformation replaces a direct memory access operation using **`ioread32be()`** with a function call to **`uart_in32()`**. The change involves restructuring the parameter order and access pattern. The original code accesses a memory-mapped I/O location by adding an offset to a base address stored in the **`port`** structure (**`port->base`**). The transformed code passes the offset directly as the first argument to **`uart_in32()`** and the entire **`port`** structure as the second argument. This requires GPT-4 to correctly identify not only the function name change but also the argument reordering and structural relationship between the original address calculation and the new function parameters. Despite this complexity, GPT-4 successfully generates a valid semantic patch for this change set across all k values.

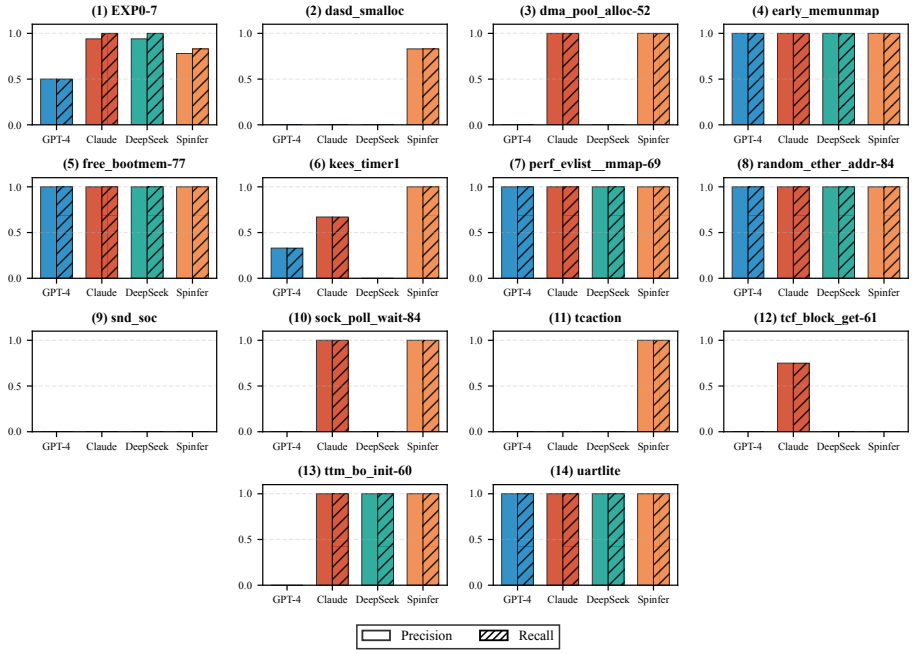

Fig. 7 Comparison of deep learning models and Spinfer across 14 test directories when using three input change examples (k=3) to infer the semantic patches.

```
@@
expression E1, E2;
@@
- early_iounmap(E1, E2)
+ early_memunmap(E1, E2)
```

Fig. 8 The semantic patch generated by GPT-4 for **`early_memunmap`**, showing a function renaming transformation.

```
@@
identifier base;
expression offset;
identifier port;
@@
- ioread32be(port->base + offset)
+ uart_in32(offset, port)
```

Fig. 9 The semantic patch generated by GPT-4 for **`uartlite`**, showing transformation from direct memory access to a function call with parameter reordering.

For **`kees_timer1`** and **`dasd_smalloc`**, GPT-4 generates valid semantic patches, but their effectiveness varies across different numbers of change examples (k). The change set **`kees_timer1`** involves code fragments with dependencies between function arguments and variable references, as shown in Figure 10. The semantic patch transforms three separate operations into a single function call, where the timer variable, function name, and data expression from the original code must be

correctly passed as arguments to the new function. This creates a data dependency relationship where all parameters must be correctly extracted and ordered. This semantic patch fails to achieve 1.00 precision and recall because it only covers 1 out of 3 variants. The other uncovered variants have different ordering of the function and data field assignments. On the other hand, **dasd_smalloc** involves code fragments with data dependencies between the modified and context lines. This change set also experience performance variations for the same reason as **kees_timer1**, which is incomplete semantic patch coverage.

```
@@
expression timer_variable, function_name, data_expression;
@@
- init_timer(&timer_variable);
- timer_variable.function = function_name;
- timer_variable.data = data_expression;
+ setup_timer(&timer_variable, function_name, data_expression);
```

Fig. 10 The semantic patch generated by GPT-4 for **kees_timer1**, where three separate operations (timer initialization and setting two fields) are unified into a single function call. This patch only covers one specific ordering pattern.

Furthermore, **EXP0-7** has an issue due to an incorrect metavariable type specification. In Figure 11, **root** should be declared as **struct btrfs_root *root;** rather than **identifier**. When **root** is typed as an **identifier**, the semantic patch will match any identifier named "root" regardless of its actual type, potentially causing unintended transformations. Although this change is straightforward and does not involve any data dependencies, the semantic patch still fails as **root** is incorrectly abstracted.

```
@@
identifier root;
@@
- root->objectid
+ root->root_key.objectid
```

Fig. 11 The semantic patch generated by GPT-4 for **EXP0-7**. A direct field access is replaced with a nested field access. The patch incorrectly uses **identifier** instead of **struct btrfs_root *** for the root variable type.

For **dma_pool_alloc-52**, GPT-4 generates a valid semantic patch that yields 0.00 precision and recall. The changes in **dma_pool_alloc-52** involve a transformation that replaces a two-step operation (memory allocation followed by zeroing) with a single function call that performs both operations. This change set involves dependencies between code fragments across the modified lines, and the semantic patch fails to match any pattern because of an incorrect constraint **when != MemPtr**. This constraint specifies that between the memory allocation **MemPtr = dma_pool_alloc(Pool, Flags, &DmaPtr);** and **memset(MemPtr, 0, ...);**, the variable **MemPtr** should not appear at all. This overly restrictive constraint prevents the pattern from matching valid code sequences where **MemPtr** might be used between these operations, such as in a NULL test.

```
@@
expression Pool, Flags, DmaPtr;
expression MemPtr;
@@
- MemPtr = dma_pool_alloc(Pool, Flags, &DmaPtr);
+ MemPtr = dma_pool_zalloc(Pool, Flags, &DmaPtr);
 ... when != MemPtr
- memset(MemPtr, 0, ...);
```

Fig. 12 The semantic patch generated by GPT-4 for **dma_pool_alloc-52**. The transformation replaces a separate allocation and a zeroing operation with a single function. The patch fails due to an overly restrictive constraint **when != MemPtr**.

The semantic patch for **ttm_bo_init-60** is also valid but yield precision and recall 0.00 due to improper abstraction of the function arguments. The changes in **ttm_bo_init-60** eliminates a **NULL** parameter from the original function call while preserving all other arguments in their respective positions, as illustrated in Figure 13. The semantic patch fails because there is a mismatch in the number of function arguments. While the intended pattern should match a function call with 12 arguments with detailed expressions for each position, the generated semantic patch only has three arguments (**args_before**, **NULL**, and **args_after**). This change set involves code fragments with dependencies between the removed and added lines, as shown in Figure 13.

```
@@
identifier func = ttm_bo_init;
expression args_before, args_after;
@@
- func(args_before, NULL, args_after)
+ func(args_before, args_after)
```

Fig. 13 The semantic patch generated by GPT-4 for **ttm_bo_init-60**. The generated patch oversimplifies the argument structure by using generic metavariables (**args_before** and **args_after**) instead of precisely matching the 12 specific arguments in the original function call.

The remaining change sets **snd_soc**, **sock_poll_wait-84**, **tcf_block_get-61**, and **dma_pool_alloc-52** yield 0.00 precision and recall due to invalid semantic patches. The change sets **snd_soc**, **sock_poll_wait-84**, and **tcf_block_get-61** involve code fragments with dependencies between the modified and context lines, while dependencies of code fragments in **dma_pool_alloc-52** are only between the added and removed lines. The failures in **snd_soc**, **sock_poll_wait-84**, **tcf_block_get-61** stem from metavariables appearing only in the added lines. This causes Coccinelle to lack sufficient information about what terms to construct for these metavariables. Figure 14 illustrates this problem with **sock_poll_wait-84**, where the semantic patch introduces a new parameter **sock** that is not present in the original function call, but fails to properly define it in the preceding context or previously removed lines. Similarly, the failure in **dma_pool_alloc-52** is due to the appearance of dot notation (...) in the added line.

Furthermore, the semantic patch generated by GPT-4 for **tcaction** fails due to improper metavariable definitions. Figure 15 shows output from one trial where it

```
@@
identifier f;
expression sock, wait;
@@
- sock_poll_wait(f, wait)
+ sock_poll_wait(f, sock, wait)
```

Fig. 14 The semantic patch for modifying **sock_poll_wait** function calls, which fails due to the introduced metavariable **sock** only appearing in the added line and lacking proper context definition.

lacks the required declarations of **list_for_each_entry** and **tcf_exts_for_each_action** as **iterator name**. In another trial, while the model defines **list_for_each_entry** as an iterator name, it does so incorrectly with **iterator list_for_each_entry;** without the keyword **name** after the keyword **iterator**. This makes **list_for_each_entry** a metavariable that can match any iterator. This change set involves code fragments with dependencies between the removed and added lines, as shown in Figure 15.

```
@@
identifier a;
expression exts;
@@
- LIST_HEAD(actions);
+ int i;
  ...
- tcf_ext_to_list(exts, &actions);
- list_for_each_entry(a, &actions, list)
+ tcf_exts_for_each_action(i, a, exts)
  { ... }
```

Fig. 15 The semantic patch generated by GPT-4 for **tcaction**. The semantic patch in invalid due to missing iterator declarations for **list_for_each_entry** and **tcf_exts_for_each_action** in the metavariable definitions.

DeepSeek achieves perfect precision and recall (1.00) across different numbers of change examples (k) for most change sets with transformations involving code fragments with dependencies between modified lines of code, as illustrated in Figure 8. These successful cases include **early_memunmap**, **free_bootmem-77**, **uartlite**, **perf_evlist__mmap-69**, **random_ether_addr-84**, and **ttm_bo_init-60**.

DeepSeek also generates valid semantic patches but with varying effectiveness across different k values for **dma_pool_alloc-52**, **sock_poll_wait-84**, **dasd_smalloc**, and **EXP0-7**. Specifically, the change set **dma_pool_alloc-52** achieves only 0.50 precision and recall at k=2, and dropping to 0.00 for k={1,3}. This lower performance at k={1,2} is due to incorrect context identification as shown in Figure 12, and invalid semantic patch at k=3 due to metavariables appearing only in the added lines, similar to Figure 14. Next, **sock_poll_wait-84** achieves perfect scores at k=1 but drops to 0.00 for k={2,3} because the semantic patch is invalid due to metavariables appearing only in the added lines. For **dasd_smalloc**, precision and recall reach 0.75 at k={1,2} because one variant remains uncovered by the generated semantic patches, while at k=3, the semantic patch is invalid due to unintended characters appears in the semantic patch. Moreover, **EXP0-7** shows perfect

results at k=1, but fails to achieve precision and recall of 1.00 at k={2,3} due to an incorrect metavariable type, similar to the error in Figure 11.

DeepSeek consistenly fail to yield valid semantic patches across all number of change examples (k) for **kees_timer1**, **snd_soc**, **tcaction**, and **tcf_block_get-61**. For **snd_soc**, **tcaction**, and **tcf_block_get-61**, the primary cause is metavariables appearing only in the added lines, similar to the issue in Figure 14. In addition, the failure in **kees_timer1** stems from improper metavariable definitions. In Figure 16, the timer structure contains both **function** and **data** fields. The semantic patch fails due to **function** and **data** being used as both field names and metavariables of the assigned values.

```
1 @@
2 expression timer, function, data;
3 @@
4 - init_timer(&timer);
5 - timer.function = function;
6 - timer.data = (unsigned long)data;
7 + setup_timer(&timer, function, (unsigned long)data);
```

Fig. 16 The semantic patch generated by GPT-4 for **kees_timer1**. This semantic patch fails due to incorrect metavariable definition. The **function** and **data** names are used as both field names and metavariables of the assigned values.

Claude consistently achieves precision and recall of 1.00 for the following change sets: **dma_pool_alloc-52**, **early_memunmap**, **free_bootmem-77**, **perf_evlist__mmap-69**, **random_ether_addr-84**, **ttm_bo_init-60**, and **uartlite**. Conversely, Claude consistently yields 0.00 precision and recall for **dasd_smalloc** and **snd_soc** because of invalid semantic patches due to metavariables appearing only in the added lines, similar to the example shown in Figure 14.

The semantic patches generated by Claude show varied performance across different numbers of change examples (k) for **sock_pool_wait-84**, **tcf_block_get-61** , **EXP0-7**, **kees_timer1**, and **tcaction**. For **sock_pool_wait-84**, the precision and recall are 0.00 for k={1,2} because the generated semantic patches are invalid due to metavariables incorrectly appearing only in the added lines. However, the precision and recall improve to 1.00 at k=3. Similarly, **tcf_block_get-61** shows precision and recall of 0.00 at k=1 for the same reason, then improves to 1.00 at k=2. The precision and recall drops to 0.75 at k=3 due to missing context lines. We leverage Figure 17 to exemplify this case. At k=2, Claude successfully includes the function signature in line 16 that contains the type information for the argument **extack** in line 19, which is **struct netlink_ext_ack***. the model incorrectly specifies the **extack** argument in line 19 as an **expression** metavariable instead of including the type information from the function signature in line 16. Consequently, the second rule to always be applied after the first rule. For **EXP0 -7**, **kees_timer1**, and **tcaction**, the results remain inconsistent across different k values primarily due to either semantic patches not covering all variants, incorrect metavariable type specifications, or invalid semantic patches.

Spinfer successfully achieves 1.00 precision and recall across different k values for **dma_pool_alloc-52**, **early_memunmap**, **free_bootmem-77**, **kees_timer1**, **tcaction**, **perf_evlist__mmap-69**, **random_ether_addr-84**, **sock_poll_wait-84**, **ttm_bo_init-60**. Among these, **sock_poll_wait-84** involves code fragments with dependencies be-

```
@add_sch@
expression block, filter_list;
identifier fn;
@@
fn(...) {
<...
- tcf_block_get(block, filter_list)
+ tcf_block_get(block, filter_list, sch)
...>
}

@add_extack depends on add_sch@
expression block, filter_list;
identifier fn;
@@
fn(..., struct netlink_ext_ack *extack, ...) {
<...
- tcf_block_get(block, filter_list, sch)
+ tcf_block_get(block, filter_list, sch, extack)
...>
}
```

Fig. 17 The semantic patch generated by Claude for **`tcf_block_get-61`** at k=2. This semantic patch demonstrates the correct context identification where the last function parameter of **`tcf_block_get`** is properly derived from the function signature.

tween the modified and context lines, while the dependencies of code fragments in the remaining change sets are only between the added and removed lines.

For **`dasd_smalloc`**, Spinfer initially achieves precision and recall of 0.75 at k=1 and k=2. This improves to 0.83 for both metrics at k=3. The imperfect precision and recall is because the semantic patch does not cover all change variants in the input change examples. Moreover, **`uartlite`** initially shows 0.00 precision and recall at k=1 because Spinfer fails to produce any semantic patches when given a single change example. However, the performance improves to 1.00 for both precision and recall at k=2 and k=3.

Spinfer fails to produce semantic patches for **`snd_soc`** and **`tcf_block_get-61`**. One common failure reason for these directories is the parameter lists are not all the same in all the change examples. Figure 18 illustrates this case, where the parameter list in line 2 and line 9 is different. The needed parameter is always the last one **`struct snd_soc_dai *dai`**, but Spinfer fails to make the generalization. This happens for all k values in **`snd_soc`** and at k=3 for **`tcf_block_get-61`**.

RQ1 Answer: Spinfer outperforms deep learning models in generating valid semantic patches with higher precision and recall on the inference split. All approaches perform well on simple transformations involving code fragments with no dependencies and with dependencies between the modified lines, while struggling to handle dependencies between modified and context lines.

4.2 RQ2: How well do the semantic patches generated by deep learning models and Spinfer generalize to unseen code?

We evaluate the generalization capability of semantic patches to C code not present in the input examples. This assessment reflects real-world scenarios where patches must handle varying code patterns. Our methodology applies generated patches

```
1 // Change 1 from Variant 1
2 static int mt2701_dlm_fe_startup(struct snd_pcm_substream *substream, struct snd_soc_dai
      *dai)
3 {...
4 + struct mtk_base_afe *afe = snd_soc_dai_get_drvdata(dai);
5 ...}
6
7 // Change 1 from Variant 2
8 static int mtk_afe_fe_trigger(struct snd_pcm_substream *substream, int cmd, struct
      snd_soc_dai *dai)
9 {...
10 + struct mtk_base_afe *afe = snd_soc_dai_get_drvdata(dai);
11 ...}
```

Fig. 18 The case where the change examples in **snd_soc** has different parameter lists. Both changes add identical **struct mtk_base_afe *afe** variable declarations using **snd_soc_dai_get_drvdata(dai)**, but occur in the functions with different parameter lists. Change 1 from Variant 1 has two parameters while Change 1 from Variant 2 has three parameters.

to unseen code C_{in} from the test split, producing transformed code C_{out}. We then compare C_{out} with ground truth transformations C_{truth} using precision and recall metrics. The evaluation is conducted using different numbers of input examples (k={1, 2, 3}). First, we examine the global-level patch validity, precision, and recall for each approach in Section 4.2.1. Then, we present and discuss the performance per change set in Section 4.2.2.

4.2.1 Global Performance

Table 4 compares the performance of different approaches in generating semantic patches that generalize to unseen code. We measure the precision and recall in two settings: all and valid-only. In the all setting, we incorporate all change sets in our calculations, including those with invalid or no semantic patches. We assign the precision and recall values to 0.00 for these change sets. In the valid-only setting, we only consider change sets with valid semantic patches when computing the metrics, which provides insight into the quality of successfully generated patches independent of failure cases. This dual reporting approach allows us to evaluate both the models' overall effectiveness across all change sets and their performance specifically on cases where they successfully generate valid semantic patches.

GPT-4 and DeepSeek show declining performance with additional examples when considering all change sets. With k=1, GPT-4 achieves precision and recall of 0.45, which decreases to 0.35 for both metrics with k=2, and shows a slight increase to 0.36 for both metrics with k=3. However, when excluding change sets with invalid semantic patches, GPT-4 maintains relatively consistent performance across different k values, with precision and recall around 0.71-0.74. DeepSeek similarly exhibits decreasing performance with additional examples when including all change sets. Starting with 0.53 for both precision and recall at k=1, the metrics decline to 0.39 at k=2, and remain at 0.38 for k=3. Similar to GPT-4, DeepSeek's performance when excluding change sets with invalid semantic patches is significantly higher, achieving 0.90-0.94 precision and recall across all k values, suggesting that its main limitation is producing valid semantic patches.

Claude and Spinfer demonstrate different performance patterns as the number of change examples increases. Claude start with precision and recall of 0.47 at k=1

Table 4 Precision and recall comparison of GPT-4, Claude, DeepSeek, and Spinfer on the test split, measuring how well generated semantic patches generalize to unseen C code. **All**: Change sets with invalid semantic patches or no semantic patches are included in calculations. **Valid-only**: Change sets with invalid semantic patches or no semantic patches are excluded from calculations.

#Examples (k)	Approach	Precision		Recall	
		All	Valid-only	All	Valid-only
1	GPT-4	0.43	0.72	0.44	0.74
	Claude	0.45	0.93	0.45	0.93
	DeepSeek	0.51	0.90	0.51	0.90
	Spinfer	0.16	0.23	0.16	0.23
2	GPT-4	0.34	0.73	0.34	0.73
	Claude	0.60	0.83	0.60	0.83
	DeepSeek	0.38	0.94	0.38	0.94
	Spinfer	0.46	0.66	0.46	0.65
3	GPT-4	0.35	0.71	0.35	0.71
	Claude	0.56	0.85	0.56	0.85
	DeepSeek	0.36	0.91	0.36	0.91
	Spinfer	0.49	0.70	0.49	0.70

when including all change sets. Subsequently, the precision improves to 0.57 at k=2 (though recall drops to 0.35), and both metrics increase to 0.62 for k=3, showing improvement with more examples. When excluding the change sets with invalid semantic patches, Claude achieves the highest precision among all approaches at k=1 with 0.93, though this decreases slightly to 0.83-0.85 for k=2 and k=3. Spinfer starts with the lowest performance of 0.16 for both metrics at k=1 when including all change sets, then shows significant improvement to 0.47 at k=2, and further increases to 0.50 at k=3. This improvement is also reflected when excluding change sets with no semantic patches produced, where the performance rises from 0.23 at k=1 and 0.70 at k=3. The low performance of Spinfer at k=1 is due to its inability to abstract code fragments into metavariables when provided with only a single example.

Overall, the deep learning models demonstrate strong performance when generating semantic patches from just a single change example, particularly when considering only the valid semantic patches they produce. In contrast, Spinfer is specifically designed to improve with more change examples. Leveraging more change examples incorporate more variation and this helps Spinfer to create more generic and widely applicable rules. However, if Spinfer's clustering mechanism does not operate at the right granularity, additional change examples might actually lead to overfitted rules that are too specific.

4.2.2 Change Set Performance

Figures 19, 20, and 21 show the performance for different change sets in the test split with different numbers of change examples (k). Table 5 summarizes which approaches perform best in each change set. Our analysis in this section focuses primarily on the change sets that achieved perfect precision and recall (1.00) in the RQ1 experiment, as discussed in Subsection 4.1.2. The change sets that did not

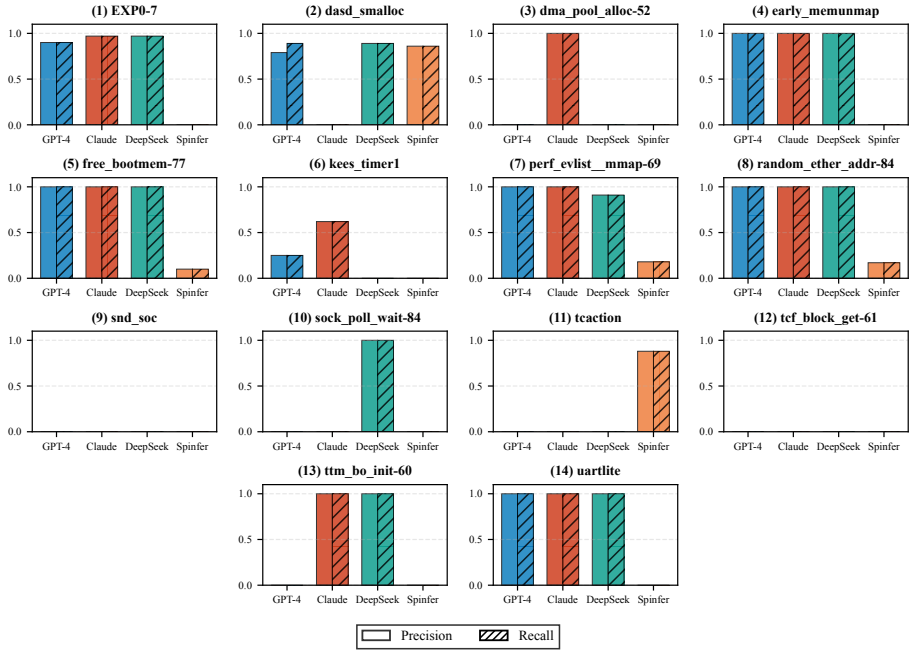

Fig. 19 The generalization performance comparison of deep learning models and Spinfer across 14 test change sets when using one input change example (k=1) to infer the semantic patches.

Change Set	k=1	k=2	k=3
EXP0-7	C/D	G/C/D	G/C/D
dasd_smalloc	S	D	G
dma_pool_alloc-52	C	S	S
early_memunmap	C/D/G	All	All
free_bootmem-77	C/D/G	All	All
kees_timer1	S	C	C
perf_evlist__mmap-69	D/G	D	D
random_ether_addr-84	C/D/G	G/C/D	All
snd_soc	-	G	G
sock_poll_wait-84	S	S	C/S
tcaction	S	S	S
tcf_block_get-61	C	C	C
ttm_bo_init-60	C	C	D
uartlite	C/D/G	All	All

Table 5 Best performing model for each change set and number of examples (k). G=GPT-4, C=Claude, D=DeepSeek, S=Spinfer, All=All models perform equally, -=No model performs well. Performance comparison based on combined precision and recall metrics.

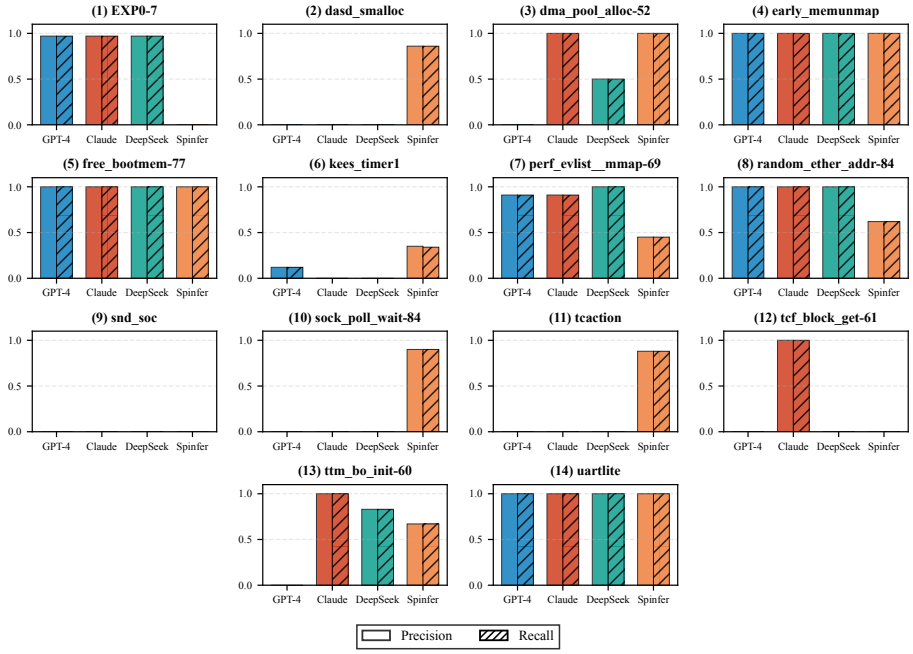

Fig. 20 The generalization performance comparison of deep learning models and Spinfer across 14 test change sets when using two input change examples (k=2) to infer the semantic patches.

achieve perfect precision and recall (1.00) will exhibit the same problems identified in Section 4.1.2 and are therefore excluded from this analysis to avoid redundancy.

The semantic patches generated by GPT-4 achieves precision and recall of 1.00 on both inference and test splits across all k values for **`early_memunmap`**, **`free_bootmem-77`**, **`random_ether_addr-84`**, and **`uartlite`**. These changes involve code fragments with dependencies across the modified lines. Moreover, **`perf_evlist__mmap-69`** consistently achieves perfect precision and recall on the inference split but fails on the test split. Figure 22 shows the semantic patch generated by GPT-4 for this change. The intended transformation should remove the third argument regardless of its value, but GPT-4's implementation specifically targets the removal of the literal **`false`**. This failure occurs because the third argument is consistently **`false`** across all change examples provided during patch inference. Consequently, GPT-4 assumes this argument should not be abstracted.

DeepSeek generates semantic patches that achieve perfect precision and recall (1.00) on the inference and test splits across different k values for **`early_memunmap`**, **`free_bootmem-77`**, **`perf_evlist__mmap-69`**, **`random_ether_addr-84`**, and **`uartlite`**. Notably, DeepSeek achieves perfect precision and recall for **`perf_evlist__mmap-69`**, where GPT-4 falls short. The changes involves code fragments with dependencies across the modified lines. For **`ttm_bo_init`**, which initially achieves perfect scores on the inference split, the performance drops on the test split. At k=1 and k=3, both precision and recall are 1.00. At k=2, it drops to 0.83. This lower performance

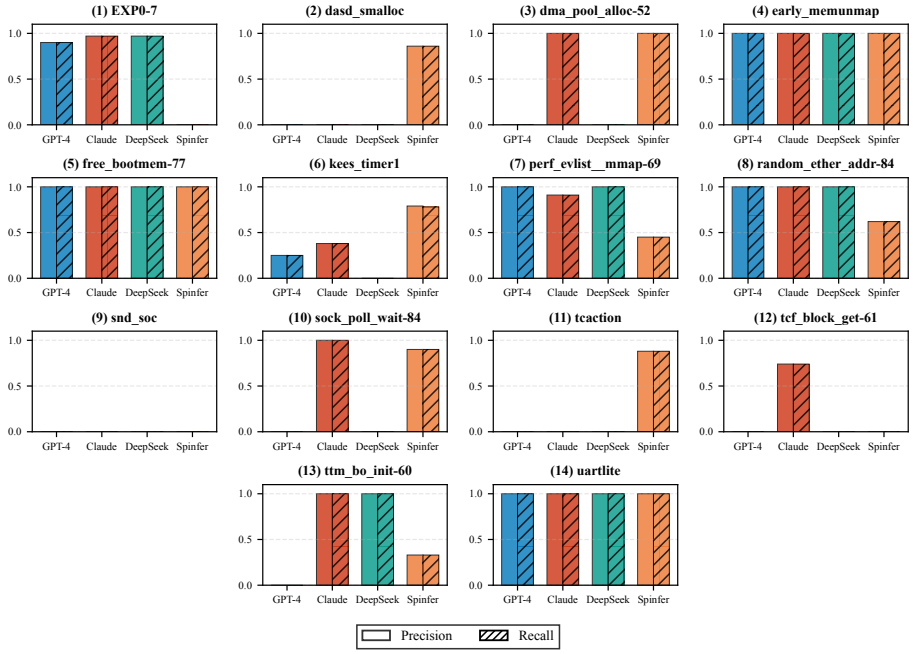

Fig. 21 The generalization performance comparison of deep learning models and Spinfer across 14 test change sets when using three input change examples (k=3) to infer the semantic patches.

```
1  // The Semantic patch
2  @@
3  expression E1, E2;
4  @@
5  - perf_evlist__mmap(E1, E2, false)
6  + perf_evlist__mmap(E1, E2)
7
8  // Change Example 1
9  - err = perf_evlist__mmap(evlist, 128, false);
10 + err = perf_evlist__mmap(evlist, 128);
11
12 // Change Example 2
13 - err = perf_evlist__mmap(evlist, opts.mmap_pages, false);
14 + err = perf_evlist__mmap(evlist, opts.mmap_pages);
```

Fig. 22 The semantic patch for **perf_evlist__mmap-69** generated by GPT-4. The last argument should be abstracted into an expression metavariable.

occurs because one of the arguments is not abstracted to a metavariable. This is similar to the issue illustrated in Figure 22

Claude successfully generates semantic patches that achieve perfect precision and recall on both inference and test splits in all different change example count (k) for **dma_pool_alloc-52**, **early_memunmap**, **free_bootmem-77**, **random_ether_addr-84**, **ttm_bo_init-60**, and **uartlite**. All these change sets involve code fragments with dependencies across the modified lines. Similar to GPT-4, Claude also faces difficulty with **perf_evlist__mmap-69** as it achieves perfect precision and recall on

the inference split but fails on the test split because one of the arguments is not abstracted to a metavariable. This is similar to the issue illustrated in Figure 22.

For Spinfer, at k=1, none of the semantic patches that previously achieves precision and recall 1.00 generalize to the test split. The performance at k=1 is very low due to Spinfer's inability to abstract code fragments into metavariables when provided with only a single example. For k=2 and k=3, the semantic patches for **`dma_pool_alloc-52`**, **`early_memunmap`**, **`free_bootmem-77`**, and **`uartlite`** consistently achieve perfect precision and recall (1.00) on both inference and test splits across different k values. All these change sets involve code fragments with dependencies across the modified lines.

On the other hand, the semantic patches generated by Spinfer for **`kees_timer1`**, **`perf_evlist__mmap-69`**, **`random_ether_addr-84`**, **`tcaction`**, and **`ttm_bo_init-60`** achieve perfect precision and recall on the inference split but fail on the test split. All these change sets involve code fragments with dependencies across different the modified lines. The main reason for the failure in these change sets is Spinfer's inability to properly convert code fragments into metavariables, as illustrated in Figure 22, so it behaves like GPT-4. The change set **`sock_poll_wait-84`** also results in failure on the test split, even though the precision and recall are 1.00 on the inference split. The failure reason is because Spinfer fails to abstract the context lines properly.

RQ2 Answer: Semantic patches generalize differently across approaches. Claude performs the best overall. Spinfer struggles with a single example but improves with additional examples, while GPT-4 and DeepSeek show declining performance as the number of input examples increases. The main challenge for all approaches is properly abstracting code fragments into metavariables. Simpler changes generally achieve better generalization across all models than those with dependencies across different lines of code.

4.3 RQ3: How do variations in the input change examples affect the performance of semantic patches generated by deep learning models and Spinfer on unseen code?

We investigate how each approach handles change sets with different example diversities by distinguishing between single-variant and multi-variant change sets. Single-variant change sets contain modifications that follow identical patterns of removals and additions, while multi-variant change sets contain several changes that achieve the same semantic goal but differ in control flow, ordering, or structure (e.g., **`kees_timer`** in Figure 1). Figure 23 presents the experimental results across different numbers of examples.

GPT-4 shows precision and recall of 0.56 for single-variant transformations with one change example (k=1), and these metrics remain stable as the number of change examples increase, with 0.55 at k=2 and 0.56 at k=3. For multi-variant change sets, GPT-4 achieves precision and recall of 0.39/0.41 at k=1, but the performance decreases to 0.22 at k=2 and 0.23 at k=3.

Claude demonstrates strong performance, particularly with single-variant change sets. For the single-variant change sets, Claude achieves precision and recall of 0.78 with a single example (k=1), which remains stable at 0.77 for k=2 and improves to 0.88 with k=3. For multi-variant, Claude shows steady improvement as the

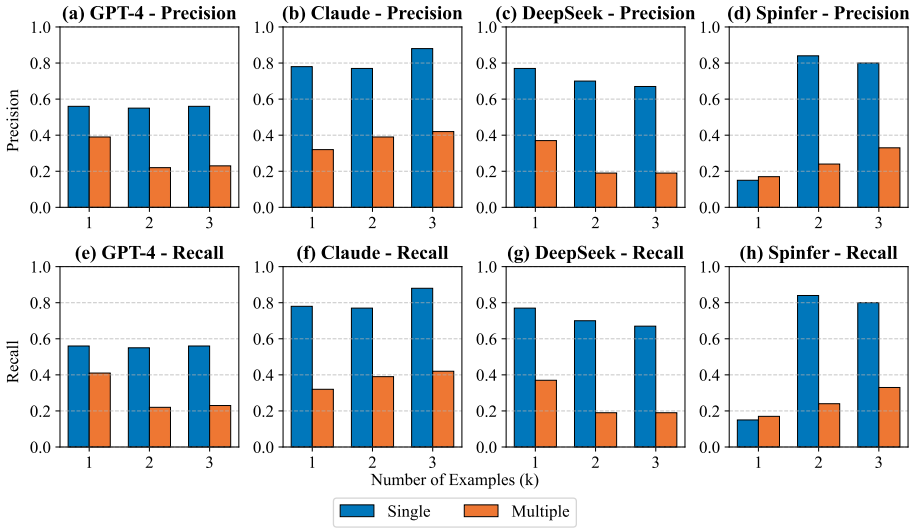

Fig. 23 The precision and recall comparison of GPT-4, Claude, DeepSeek, and Spinfer on unseen contrasting the performance between change sets containing single-variant against multi-variant. The x-axis is the number of change examples (k) used to generate the semantic patches during the patch inference phase.

number of change examples increases, with precision and recall of 0.32 at k=1. Both metrics improve to 0.39 at k=2, and further to 0.42 at k=3.

DeepSeek performs well on the single-variant change sets but shows inconsistent results on the change sets containing multiple variants. It achieves precision and recall of 0.77 for the single-variant case, leveraging a single change example (k=1). Both metrics declines to 0.70 at k=2 and further to 0.67 at k=3. For multi-variant change sets, the precision and recall are 0.37 with one change example and drops considerably to 0.19 with k={2, 3}.

Spinfer exhibits a different performance trend than the deep learning models. With one example (k=1), Spinfer achieves only 0.15 precision and recall on the single-variant change sets and 0.17 on the multi-variant change sets. However, with k=2, Spinfer's performance improves dramatically to 0.84 for the single-variant change sets, and still maintains strong performance at 0.80 at k=3. For the multi-variant change sets, Spinfer shows steady improvement as the number of change examples (k) increases, from 0.17 at k=1 to 0.24 at k=2 and 0.33 at k=3.

Our experiments reveal several notable trends across models. First, except for Spinfer at k=1, all models perform better on single-variant directories than multi-variant directories. Second, performance patterns with additional examples vary significantly between models: Claude and Spinfer generally show consistent improvement, while GPT-4's performance remains stable for the single-variant case but declines for the multi-variant case, and DeepSeek shows declining performance as the number of examples increases for both the single-variant case and the multi-variant case. Third, the performance gap between single and multi-variant change sets persists across all models and example counts, highlighting the inherent com-

plexity of generating semantic patches that generalize effectively across diverse change variants.

RQ3 Answer: Overall, all approaches perform better on single-variant change sets than multi-variant change sets, with each model responding differently to the presence of additional examples. The performance gap between single-variant and multi-variant change sets across all models highlights the inherent challenge of generating semantic patches that can effectively generalize across diverse change variants.

5 Error Analysis for RQ4

We conducted systematic error analysis on all failure cases to identify the strengths and weaknesses of each approach. Following grounded theory methodology, we analyzed failures using a bottom-up approach. For each case, we debugged the generated semantic patches to determine why they failed to produce the expected transformations. This process yielded initial error categories. Subsequently, we refined these categories by identifying common patterns across failures and developed an error categorization. We performed this analysis at k=1 and k=3, representing the worst and best cases in our experimental settings. In total, we analyzed 1,118 failure cases across 4 approaches. Figure 24 shows the number of failure instances in each model for different numbers of examples (k).

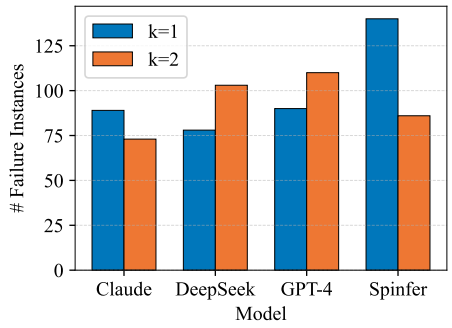

Fig. 24 The number of failure instances in each model for different number of examples (k).

Figure 25 presents our categorization of failure cases. We identified three main error types: incomplete semantic patch coverage, abstraction problems, and invalid semantic patch. We explain each category briefly as follows.

- Incomplete semantic patch coverage occurs when the generated semantic patches fail to cover all change variants from the input change examples.
- Abstraction problems arise when semantic patches either overfit or overabstract code fragments. This category can be divided further into three subcategories:
  - Under-specification, which falls into two subcategories: Missing metavariables and extra context constraints. Missing metavariables occurs when

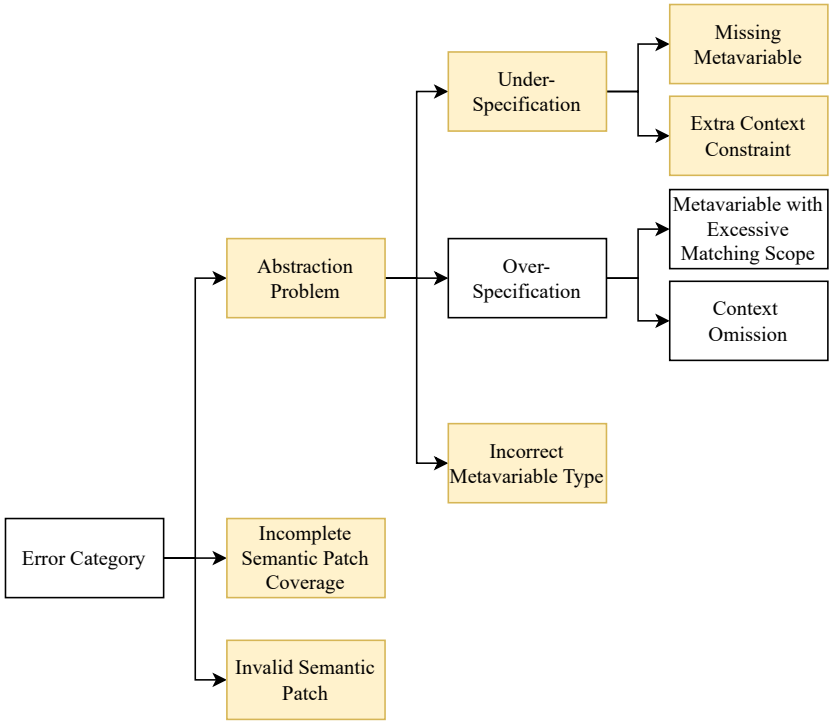

Fig. 25 Error categorization of failure cases. Yellow boxes indicate error categories common to both deep learning models and Spinfer, while white boxes represent categories exclusive to deep learning models. No error categories are unique to Spinfer.

code fragments remain concrete despite requiring abstraction. Extra context constraints occurs when patches include irrelevant context from change examples.
  - Over-specification, which includes: metavariables with excessive matching scope and context omission. Metavariables with excessive matching scope occurs when models excessively abstract code fragments. Context omission occurs when semantic patches fail to abstract the context between modified lines correctly.
  - Invalid metavariable type occurs when models specify wrong abstraction levels in metavariable definitions.
- Invalid semantic patch results primarily from grammar violations of the Semantic Patch Language (SmPL).

We discuss the error distribution of each model in Section 5.1. Then, we explain each error category and provide an example in Section 5.2.

5.1 Error Distribution in Each Model

Figure 26 shows the error distribution for each model. We computed relative distribution by dividing the number of occurrences of each error type by the total number of errors for that model.

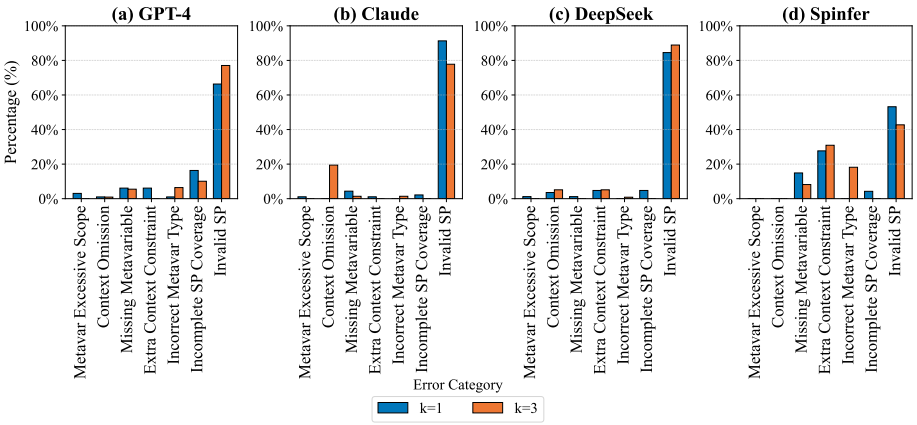

Fig. 26 The error distribution for each model across different number of change examples (k). SP stands for Semantic Patch.

The three deep learning models (GPT-4, Claude, and DeepSeek) predominantly exhibit failures due to invalid semantic patches across at k={1, 3}. GPT-4's errors with k=1 are dominated by invalid semantic patches errors (66.33%), followed by incomplete semantic patch coverage errors (16.33%). When k increases to 3, invalid semantic patch errors increase to 77.06%, while incomplete semantic patch coverage errors decrease to 10.09%. Claude shows different patterns, with invalid semantic patch errors constituting 91.30% at k=1, followed by missing metavariable errors (4.35%). At k=3, invalid semantic patch errors decrease to 77.78%, while context omission errors increase to 19.44%. DeepSeek exhibits yet another error pattern. With k=1, invalid semantic patch errors account for 84.52% of all errors, with extra context constraint errors at 4.76%. At k=3, invalid semantic patch errors increase slightly to 88.89%, with context omission errors at 5.1%.

Spinfer exhibits a diverse range of error patterns that mainly fall within underspecification, incorrect metavariable type, incomplete semantic patch coverage, and invalid semantic patch categories. Spinfer's failures primarily consist of invalid semantic patch errors (53.19%) at k=1, followed by extra context constraint errors (27.66%) and missing metavariable errors (14.89%). At k=3, invalid semantic patch errors decrease to 42.73%, while extra context constraint errors slightly increase to 30.91%. Notably, incorrect metavariable type errors emerge at k=3 (18.2%), while missing metavariable errors decrease to 8.2%. This shift suggests that while Spinfer benefits from additional examples in terms of patch validity, it still tends to produce either overly specific patches or encounters challenges with proper metavariable typing when leveraging multiple examples to infer semantic patches.

RQ4 Answer: Deep learning models predominantly fail due to invalid semantic patch errors. Spinfer ensures syntactic correctness but creates overly specific patches with unnecessary constraints. While additional examples generally reduce some error types, they sometimes introduce new problems, as in Claude and Spinfer.

5.2 Explanation and Example of Each Error Category

5.2.1 Under-Specification

Under-specification occurs when models do not properly abstract certain code fragments, making generated semantic patches too specific. This error affects both Spinfer and deep learning models, and is divided into missing metavariables and extraneous context constraints.

Missing metavariables occur when code fragments remain concrete despite requiring abstraction. This happens when change examples contain common code fragments that should be abstracted to metavariables, but the values are the same in all input change examples. Figure 22 showed an example for this error case, where the goal is to remove the third argument of the function `perf_evlist__mmap`. This third argument can be any expressions. However, the third argument in all the change examples used to infer the semantic patch is `false`. Consequently, the model infers that this third argument should not be abstracted.

Extra context constraints refer to including unnecessary surrounding code fragments (like function signatures or entire assignment statements) from the input change examples. Figure 12 illustrated this error category. The semantic patch fails because the constraint `when != MemPtr` is too specific. This incorrectly requires that between the allocation statement `MemPtr = dma_pool_alloc(Pool, Flags, &DmaPtr);` and the zeroing statement `memset(MemPtr, 0, ...);`, the variable `MemPtr` should not appear at all. Such an overly restrictive constraint prevents the pattern from matching valid code sequences where `MemPtr` might be used in a null test between these operations.

5.2.2 Over-Specification

Over-specification occurs when models abstract code fragments that should remain concrete into metavariables. This category primarily affects deep learning models, and it divides into metavariables with excessive matching scope and context omission.

Metavariables with excessive matching scope occurs when models excessively abstract code fragments. This error can cause unintended transformations by applying changes to code where they should not be applied. Figure 27 illustrates this issue in GPT-4 for `dasd_smalloc`. In this example, the function name `dasd_smalloc_request` should not be abstracted to a metavariable. This inappropriate abstraction means the semantic patch would match any function with 4 arguments and add NULL, rather than specifically targeting the `dasd_smalloc_request` function as intended.

Context omission occurs when semantic patches fail to correctly include the context from the input change examples. Figure 28 shows an example of context

```
@@
identifier dasd_smalloc_request;
identifier device;
expression type, count, size;
@@
- dasd_smalloc_request(type, count, size, device)
+ dasd_smalloc_request(type, count, size, device, NULL)
```

Fig. 27 The semantic patch generated by GPT-4 for **dasd_smalloc**. The function name **dasd_smalloc_request** should not be abstracted into a metavariable.

omission in Claude's generated semantic patch for **tcf_block_get-61** at k=3. The semantic patch correctly identifies the need to add parameters to the **tcf_block_get** function calls but fails to properly incorporate the contextual information about function signatures. Specifically, in **rule2** (lines 6-10), it does not recognize that the **extack** parameter should only be added when it is available in the enclosing function signature (**struct netlink_ext_ack *extack**). The consequence of this context omission is that rule 2 will always be applied after rule 1, regardless of whether the function signature contains the necessary **extack** parameter. This contrasts with the correct implementation shown in Figure 17, where the patch properly incorporate the presence of the **extack** parameter in the function signature before applying the transformation.

```
@rule1@
expression block, filter_list;
@@
- tcf_block_get(block, filter_list)
+ tcf_block_get(block, filter_list, sch)

@rule2@
expression block, filter_list, sch;
@@
- tcf_block_get(block, filter_list, sch)
+ tcf_block_get(block, filter_list, sch, extack)
```

Fig. 28 The semantic patch generated by Claude at k=3 for **tcf_block_get-61**. This semantic patch does not include and abstract the function header **fn(..., struct netlink_ext_ack *extack,)** in **rule2** (line 6-10). The correct version is shown in Figure 17.

5.2.3 Incorrect Metavariable Type

This error occurs when models specify wrong metavariable types in the metavariable definitions. Figure 11 illustrates this error category. In this example, **root** is incorrectly declared as **identifier** when it should be declared as **struct btrfs_root *root;**. This metavariable typing error can misrepresent the nature of the code fragment being transformed. When **root** is typed as an identifier, the semantic patch will match any identifier named "root" regardless of its actual type, potentially causing unintended transformations in unrelated code that happens to use a variable or parameter with the same name but different type.

5.2.4 Incomplete Semantic Patch Coverage

This error occurs when multiple semantically equivalent change examples with different variants are present during patch inference, but the generated semantic patch only covers some variants. Figure 10 illustrated an example of this category in GPT-4. Here, the semantic patch transforms three separate operations into a single function call, where the timer variable, function name, and data expression from the original code must be correctly passed as arguments to the new function. This semantic patch fails to achieve 1.00 precision and recall because it only covers 1 out of the 3 variants. One of the variants swaps the ordering of the function and data field assignments and the other variant does not involve the data field assignment.

5.2.5 Invalid Semantic Patch

These errors stem from violations of SmPL grammar rules. The most common violation occurs when a metavariable appears only in the added lines without appearing in the context or removed lines. Figure 14 illustrated this case. This case causes the semantic patch to not know what terms to construct when performing the change. The second most common violation involves incorrect metavariable usage, as shown in Figure 16. The remaining violations include irrelevant symbols in transformation rules.

6 Lessons Learned and Future Directions (RQ5)

6.1 Lessons Learned

Our empirical evaluation reveals several important insights about the strengths and limitations of various approaches to automated semantic patch generation. The comparative analysis between Spinfer and deep learning models (GPT-4, Claude, and DeepSeek) highlights their complementary capabilities and suggests potential directions for future work.

Performance Varies Across Change Sets. Our analysis reveals that performance varies significantly across different change sets. No single approach performs optimally across all change sets. For instance, Spinfer excels in transformations like `tcaction` where deep learning models consistently fail, while Claude uniquely handles `tcf_block_get-61` where other approaches struggle.

Distinct Failure Patterns Between Approaches. The error analysis identifies distinct failure patterns between deep learning models and Spinfer. Deep learning models primarily struggle with syntax validity. Conversely, Spinfer's challenges center on abstraction level and generalization, often creating semantic patches that are syntactically correct but either too specific (with excessive context constraints), incomplete in covering all change variants, or contain incorrect metavariable types.

Sensitivity to Variant Diversity. Both approaches demonstrate sensitivity to the variant diversity of change examples. The performance gap between single-variant and multi-variant change sets persists across all approaches. This indicates a challenge in generating semantic patches that generalize across diverse change variant patterns. This finding emphasizes the importance of example selection in practical applications and suggests that semantic patch generation tools should incorporate mechanisms to identify and address variant diversity during the patch generation process. More sophisticated preliminary classification of input change examples based on their variant characteristics could guide the patch generation process toward more accurate semantic patches that cover all change variants.

Abstraction as a Key Challenge. Abstraction represents a significant challenge for all approaches. The ability to correctly identify which code fragments should be abstracted into metavariables or dot notations versus which code fragment should remain concrete is important for creating semantic patches that are both specific enough to avoid unintended transformations and general enough to apply across diverse change patterns.

Impact of Example Quantity on Performance. The number of examples used for patch inference significantly impacts performance, but with different patterns across approaches. For deep learning models like Claude and Spinfer, additional examples can increase the precision and recall of the generated semantic patches. In contrast, additional examples can sometimes decreases performance for GPT-4 and DeepSeek. This observation suggests that example selection strategies might be more important than simply increasing the number of examples.

Importance of Context Handling. The error analysis reveals that context handling is important for accurate semantic patch generation. Transformations involving complex control flow and data flow dependencies between modified and unmodified lines prove difficult. Ideally, the generated semantic patches should correctly identify and incorporate relevant context while avoiding irrelevant context. Deep learning models show promise in this area but still struggle with consistent context identification across different transformation patterns.

6.2 Future Directions

Based on our findings, we identify several promising directions for future research in semantic patch generation.

Potential for Hybrid Systems. The complementary strengths of the different approaches suggest significant potential for hybrid systems. A combined approach could leverage Spinfer's syntactic precision with deep learning models' abstraction capabilities. For example, Spinfer could generate initial transformation patterns, while deep learning models could refine these patterns ~~to ensure syntax validity~~ and appropriate abstraction levels. Such hybrid systems could potentially overcome the limitations of individual approaches and achieve more robust performance across diverse transformation patterns.

Improved Handling of Variant Diversity. Our results highlight the need for improved handling of variant diversity. While Spinfer already implements line-level clustering to handle similar code across different change variants, future work could explore more sophisticated clustering algorithms enhanced by deep learning techniques. These advanced approaches could better capture semantic similarities beyond token matching, enabling more effective generalization across variants with different implementation structures but identical semantic goals. Deep learning models could be trained to identify code patterns that are functionally equivalent despite syntactic differences, potentially addressing the performance gap observed in multi-variant directories. Additionally, representation learning techniques could help develop more robust embeddings of code transformations that capture their semantic intent rather than just syntactic structure.

Specialized Mechanisms for Optimal Abstraction. Future work could develop specialized mechanisms for determining optimal abstraction levels from multiple examples when generating semantic patches. This could involve techniques to identify which code fragments should be converted to metavariables versus which should remain concrete, possibly through static analysis of code structure and dependencies or deep learning-based technique. Additionally, methods for automatically refining abstraction based on feedback from patch application results could iteratively improve patch quality.

Transformation-Aware Patch Generation. Transformation-aware patch generation approaches that adapt their strategies based on the characteristics of the input change examples could significantly improve performance. By classifying transformations according to their complexity, variant diversity, and contextual dependencies, such systems could select the most appropriate approach for each case. This would allow for more targeted and effective patch generation across diverse transformation patterns. Initial transformation classification could be based on syntactic features like the number of modified lines, presence of control structures, or dependency patterns between modified elements, enabling the system to choose specialized generation strategies based on transformation type.

Enhanced Example Selection and Filtering. Change example selection and filtering could enhance the quality of input examples for patch generation. By identifying and filtering out outlier change examples or normalizing variant patterns, these mechanisms could improve the diversity of input change examples and thereby increase the likelihood of generating effective semantic patches. Active learning approaches that iteratively refine change example selection based on initial patch generation results could be particularly promising. Techniques from program synthesis that prioritize diverse but semantically similar examples could be especially valuable for addressing the challenges seen in multi-variant directories.

Specialized Pre-Training for Patch Generation. Exploring pre-training and fine-tuning strategies for deep learning models specifically targeting semantic patch generation could improve their performance. Current models are not specially optimized for the semantic patch generation task, but targeted pre-training on code

transformation corpora and fine-tuning on semantic patch examples could enhance their ability to generate valid semantic patches and make appropriate abstraction decisions. Code-specific pre-training strategies that emphasize semantic relationships between code fragments could help models better understand the dependencies between modified and context lines, which was a common source of errors in our analysis.

Practical Integration into Development Workflows. Future research should investigate the practical integration of semantic patch generation tools into software development workflows. This could include interactive systems that allow developers to refine generated patches, recommendation systems that suggest appropriate transformations based on code patterns, and tooling that assists in selecting optimal change examples for patch inference. The efficiency gains from automating repetitive code transformations must be balanced against the potential risks of incorrect transformations, suggesting a need for user interfaces that present transformations with appropriate confidence indicators and explanations of the reasoning behind each change.

Explainability Techniques for Generated Patches. Exploring explainability techniques for semantic patch generation could help users understand the reasoning behind generated patches. By providing explanations for why certain code fragments were abstracted into metavariables while others were kept concrete, or why certain context constraints were included, these techniques could build user trust and facilitate manual refinement of generated patches. Explanations could highlight the specific examples or patterns that influenced each decision in the patch, helping developers understand the generation process and identify potential limitations or errors in the resulting patch.

RQ5 Answer: No single approach works optimally across all transformation patterns, with deep learning models struggling with syntax validity while rule-based approaches face challenges with abstraction. All methods perform worse on multi-variant change sets, highlighting the difficulty in creating generalizable patches. Future research directions include hybrid approaches combining rule-based syntax enforcement with deep learning-based pattern abstraction, transformation-aware systems that adapt strategies based on input characteristics, and specialized techniques for handling multi-variant transformations.

7 Threats to Validity

Internal Validity. Our study could be influenced by the choices of the models and all the hyperparameters used for the models. We minimize such threats by using multiple deep learning models and the official replication package for Spinfer. The models may also have inherent random behaviors. We set a fixed random seed for the tunable models and the lowest "temperature" for chatbot models to improve stability, and we ran the experiment under each model setting three times to reduce the threats of randomness.

External Validity. Our study could be impacted by the specificity of the datasets used. The code diffs selected for testing are only in C and a subset of a broader set of potential cases. We have consulted with a Linux expert during the construction of our test set in an attempt to minimize bias and to include a diverse range of change categories. Although the idea of our study can be applicable beyond C code in the Linux kernel, we leave more evaluation with other software projects and programming languages as future work. The evaluation results of our study may evolve over time, as the models are being improved from time to time; closed-source models may even change their models and APIs behind the scene beyond our control. We have preserved our inference results in our replication package for better replicability.

Construct Validity. Our study could be threatened by our choice of input/prompt formats and evaluation metrics for the models. The source code changes can be represented in different kinds of formats, such as edit scripts, context diffs, or unified diffs (used in our study). The formats and relevant prompts we used may not be the optimal for showing the full potential of the deep learning models. We noted that more effective metrics for evaluating the semantic correctness of generated semantic patches are needed; for now, we employed manual validation of samples to compensate for the threat. To improve the inputs/prompts, we may encode more semantic information such as data-dependency in the diffs to facilitate deep learning, and may utilize test cases to evaluate the semantic correctness of the transformation results in a way similar to many other studies on automatic program transformation and repair, which we leave for future work.

8 Related Work

As far as we know, there is no existing study on adapting deep learning models for generating semantic patches for the Linux kernel. On the other hand, there are numerous studies on adapting deep learning models for other code-related software engineering tasks and learning various kinds of code transformation rules from code change examples without using deep learning.

8.1 Deep Learning for Coding Tasks

Deep learning models have shown good performance in various code-related software engineering tasks, such as code search and generation in various domains [1, 5, 30, 34–36], API recommendation [2, 9], vulnerability detection [27, 38], program translation and transformation [3, 6, 15, 22, 25, 28] automated program repair [7, 8, 11, 14, 29, 33]. Moreover, the releases of GPT-3.5 [20] and GPT-4 [19] have further demonstrated the wide spectrum of capabilities these large language and code models possess for various kinds of tasks. On the other hand, studies also show that such models can lead to bugs in the generated code [23, 32, 37].

Our study is different and complementary to existing studies, in exploring the advantages and limitations of deep learning models for the task of semantic patch generation. In particular, the existing studies on program transformation using large models (e.g., [3, 28]) often learn to transform programs directly. In contrast

to those studies, our work on generating code transformation rules via semantic patches has multiple unique benefits: First, semantic patches facilitate reuse, eliminating the need to address each change individually when multiple changes are similar, and they can be rigorously checked by the transformation engine Coccinelle for all relevant change instances, reducing the possibility of errors that may occur when changing individual instances separately. Second, semantic patches offer clear and precise explanations for changes performed in the codebase. E.g., Linux kernel developers often include semantic patches in commit logs to precisely communicate about the pattern of large-scale changes made in the codebase.

8.2 Learning Code Transformations from Examples

Multiple approaches have been proposed to facilitate code transformation utilizing input change examples, such as LASE [17], REFAZER [24], PyEvolve [4], and Spinfer [26]. LASE develops a sequence of Abstract Syntax Tree (AST) edits based on these examples and generates a transformation rule by solving the Largest Common Subsequence problem. REFAZER utilizes a domain-specific language to represent transformation rules and performs clustering to infer a unique transformation rule for each cluster. Nonetheless, LASE and REFAZER do not take control-flow into account when deriving these rules. PyEvolve utilizes graph-based matching and specialized adaptation techniques for inferring and applying transformation rules. However, PyEvolve is specifically tailored for Python and would require substantial adaptation efforts to be applied on C. Other approaches such as Sydit [16] and GENPAT [10] primarily address single change examples.

Although many other studies on automatic program transformation (e.g., [3, 28]) and repair (e.g., [7,33]) have utilized deep learning models and share the same ultimate goal as ours to transform programs in certain ways, they are not designed to generate semantic patches or general transformation rules yet.

Changes in the Linux kernel can also be complex, having the same semantics but many different syntax variants, which may challenge transformation inference approaches. Spinfer [26] is the only transformation inference approach we know of that is designed to consider control-flow dependencies and multiple change variants for the Linux kernel when deriving semantic patches. Thus, we use Spinfer as the baseline in our evaluation.

9 Conclusion

It is important to have automated code transformation tools to help developers quickly and accurately propagate changes across large codebases such as the Linux kernel. Although tools such as Coccinelle can help make the changes, they still require user-provided transformation rules, which it is better to be automatically generated as well. In this paper, we have presented a comprehensive evaluation of deep learning models and Spinfer for automatic semantic patch generation. Our experiments on real-world Linux kernel code changes provide valuable insights into the capabilities and limitations of each approach. The results demonstrate that no single approach works optimally across all transformation patterns, with each method exhibiting distinct strengths and weaknesses. Our error analysis reveals

that deep learning models primarily struggle with syntax validity, producing a high rate of invalid semantic patches that diminishes as more examples are provided. Conversely, the challenges of Spinfer challenges center on the abstraction level and generalization, where it often creates patches that are syntactically correct but too specific. These findings suggest several promising research directions: hybrid approaches combining Spinfer's syntactic precision with deep learning's abstraction capabilities; improved handling of variant diversity through more advanced clustering techniques; and specialized mechanisms for determining optimal abstraction levels.

References

1. Bui, N.D.Q., Yu, Y., Jiang, L.: Infercode: Self-supervised learning of code representations by predicting subtrees. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021, pp. 1186–1197. IEEE (2021). DOI 10.1109/ICSE43902.2021.00109
2. Chen, C., Peng, X., Chen, B., Sun, J., Xing, Z., Wang, X., Zhao, W.: More than deep learning: post-processing for API sequence recommendation. Empir. Softw. Eng. **27**(1), 15 (2022). DOI 10.1007/S10664-021-10040-2
3. Dilhara, M., Bellur, A., Bryksin, T., Dig, D.: Unprecedented code change automation: The fusion of LLMs and transformation by example. Proc. ACM Softw. Eng. **1**(FSE), 631–653 (2024). DOI 10.1145/3643755
4. Dilhara, M., Dig, D., Ketkar, A.: PYEVOLVE: automating frequent code changes in python ML systems. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 995–1007. IEEE (2023). DOI 10.1109/ICSE48619.2023.00091
5. Du, M., Luu, A.T., Ji, B., Ng, S.: Mercury: An efficiency benchmark for LLM code synthesis. CoRR (2024). DOI 10.48550/ARXIV.2402.07844
6. Eniser, H.F., Zhang, H., David, C., Wang, M., Christakis, M., Paulsen, B., Dodds, J., Kroening, D.: Towards translating real-world code with LLMs: A study of translating to rust. CoRR (2024). DOI 10.48550/ARXIV.2405.11514
7. Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., Tan, S.H.: Automated repair of programs from large language models. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 1469–1481. IEEE (2023). DOI 10.1109/ICSE48619.2023.00128
8. Huang, K., Meng, X., Zhang, J., Liu, Y., Wang, W., Li, S., Zhang, Y.: An empirical study on fine-tuning large language models of code for automated program repair. In: 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023, pp. 1162–1174. IEEE (2023). DOI 10.1109/ASE56229.2023.00181
9. Huang, Q., Wan, Z., Xing, Z., Wang, C., Chen, J., Xu, X., Lu, Q.: Let's chat to find the APIs: Connecting human, LLM and knowledge graph through AI chain. In: 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023, pp. 471–483. IEEE (2023). DOI 10.1109/ASE56229.2023.00075
10. Jiang, J., Ren, L., Xiong, Y., Zhang, L.: Inferring program transformations from singular examples via big code. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, pp. 255–266. IEEE (2019). DOI 10.1109/ASE.2019.00033
11. Jiang, N., Liu, K., Lutellier, T., Tan, L.: Impact of code language models on automated program repair. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 1430–1442. IEEE (2023). DOI 10.1109/ICSE48619.2023.00125
12. Lawall, J., Muller, G.: Coccinelle: 10 years of automated evolution in the linux kernel. In: Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018, pp. 601–614. USENIX Association (2018). URL `https://www.usenix.org/conference/atc18/presentation/lawall`

13. Lawall, J., Palinski, D., Gnirke, L., Muller, G.: Fast and precise retrieval of forward and back porting information for linux device drivers. In: Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017, pp. 15–26. USENIX Association (2017). URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/lawall
14. Li, Y., Wang, S., Nguyen, T.N.: DEAR: A novel deep learning-based approach for automated program repair. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 511–523. ACM (2022). DOI 10.1145/3510003.3510177. URL https://doi.org/10.1145/3510003.3510177
15. Liu, F., Li, J., Zhang, L.: Syntax and domain aware model for unsupervised program translation. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 755–767. IEEE (2023). DOI 10.1109/ICSE48619.2023.00072
16. Meng, N., Kim, M., McKinley, K.S.: Sydit: creating and applying a program transformation from an example. In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, pp. 440–443. ACM (2011). DOI 10.1145/2025113.2025185
17. Meng, N., Kim, M., McKinley, K.S.: LASE: locating and applying systematic edits by learning from examples. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pp. 502–511. IEEE Computer Society (2013). DOI 10.1109/ICSE.2013.6606596
18. Muller, G., Padioleau, Y., Lawall, J.L., Hansen, R.R.: Semantic patches considered helpful. ACM SIGOPS Oper. Syst. Rev. **40**(3), 90–92 (2006). DOI 10.1145/1151374.1151392
19. OpenAI: GPT-4 technical report. CoRR **abs/2303.08774** (2023). DOI 10.48550/arXiv.2303.08774
20. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P.F., Leike, J., Lowe, R.: Training language models to follow instructions with human feedback. In: Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022 (2022). URL http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html
21. Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in linux device drivers. In: Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008, pp. 247–260. ACM (2008). DOI 10.1145/1352592.1352618
22. Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R.: Understanding the effectiveness of large language models in code translation. CoRR (2023). DOI 10.48550/ARXIV.2308.03109
23. Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R.: Lost in translation: A study of bugs introduced by large language models while translating code. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024, pp. 82:1–82:13. ACM (2024). DOI 10.1145/3597503.3639226
24. Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., Hartmann, B.: Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 404–415. IEEE / ACM (2017). DOI 10.1109/ICSE.2017.44
25. Rozière, B., Lachaux, M., Chanussot, L., Lample, G.: Unsupervised translation of programming languages. In: Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020). URL https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html
26. Serrano, L., Nguyen, V., Thung, F., Jiang, L., Lo, D., Lawall, J., Muller, G.: SPINFER: inferring semantic patches for the linux kernel. In: 2020 USENIX Annual Technical Conference, pp. 235–248. USENIX Association (2020). URL https://www.usenix.org/conference/atc20/presentation/serrano
27. Steenhoek, B., Rahman, M.M., Jiles, R., Le, W.: An empirical study of deep learning models for vulnerability detection. In: 45th IEEE/ACM International Conference on Software

Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 2237–2248. IEEE (2023). DOI 10.1109/ICSE48619.2023.00188
28. Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., Poshyvanyk, D.: On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp. 25–36. IEEE / ACM (2019). DOI 10.1109/ICSE.2019.00021
29. Wang, W., Wang, Y., Joty, S., Hoi, S.C.H.: Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, pp. 146–158. ACM (2023). DOI 10.1145/3611643.3616256
30. Wang, Y., Wang, W., Joty, S.R., Hoi, S.C.H.: CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, pp. 8696–8708. Association for Computational Linguistics (2021). DOI 10.18653/V1/2021.EMNLP-MAIN.685
31. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022 (2022). URL http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html
32. Wu, Y., Jiang, N., Pham, H.V., Lutellier, T., Davis, J., Tan, L., Babkin, P., Shah, S.: How effective are neural networks for fixing security vulnerabilities. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023, pp. 1282–1294. ACM (2023). DOI 10.1145/3597926.3598135
33. Xia, C.S., Wei, Y., Zhang, L.: Automated program repair in the era of large pre-trained language models. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 1482–1494. IEEE (2023). DOI 10.1109/ICSE48619.2023.00129
34. Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J.: A systematic evaluation of large language models of code. In: MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022, pp. 1–10. ACM (2022). DOI 10.1145/3520312.3534862
35. Yusuf, I.N.B., Jamal, D.B.A., Jiang, L.: Automating arduino programming: From hardware setups to sample source code generation. In: 20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023, pp. 453–464. IEEE (2023). DOI 10.1109/MSR59073.2023.00069
36. Yusuf, I.N.B., Jiang, L., Lo, D.: Accurate generation of trigger-action programs with domain-adapted sequence-to-sequence learning. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 99–110. ACM (2022). DOI 10.1145/3524610.3527922
37. Zhang, Q., Fang, C., Yu, B., Sun, W., Zhang, T., Chen, Z.: Pre-trained model-based automated software vulnerability repair: How far are we? IEEE Trans. Dependable Secur. Comput. **21**(4), 2507–2525 (2024). DOI 10.1109/TDSC.2023.3308897
38. Zhou, X., Zhang, T., Lo, D.: Large language model for vulnerability detection: Emerging results and future directions. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE 2024, Lisbon, Portugal, April 14-20, 2024, pp. 47–51. ACM (2024). DOI 10.1145/3639476.3639762