PORTGPT: Towards Automated Backporting Using Large Language Models

Zhaoyang Li[†*], Zheng Yu[‡*], Jingyi Song[†], Yuxuan Luo[§], Meng Xu[¶], Dongliang Mu[†]
[†]Huazhong University of Science and Technology
{*lizy04, jingyisong, dzm91*}*@hust.edu.cn*
[‡]Northwestern University, [§]Canonical Ltd., [¶]University of Waterloo
zheng.yu@northwestern.edu, yuxuan.luo@canonical.com, meng.xu.cs@uwaterloo.ca

***Abstract*—Patch backporting, the process of migrating mainline security patches to older branches, is an essential task in maintaining popular open-source projects (e.g., Linux kernel). However, manual backporting can be labor-intensive, while existing automated methods, which heavily rely on predefined syntax or semantic rules, often lack agility for complex patches.**

In this paper, we introduce PORTGPT, an LLM-agent for end-to-end automation of patch backporting in real-world scenarios. PORTGPT enhances an LLM with tools to access code on-demand, summarize Git history, and revise patches autonomously based on feedback (e.g., from compilers), hence, simulating human-like reasoning and verification. PORTGPT achieved an 89.15% success rate on existing datasets (1815 cases), and 62.33% on our own dataset of 146 complex cases, both outperforms state-of-the-art of backporting tools. We contributed 9 backported patches from PORTGPT to the Linux kernel community and all patches are now merged.

1. Introduction

Large-scale open source projects (*e.g.,* Linux kernel, Node.js, Debian, PostgreSQL, Kubernetes) tend to maintain mainline, stable, and Long-Term Support (LTS) branches to ensure stability, continuous feature delivery, and long-term maintenance [41], [43]. When bugs are discovered in a project, developers tend to fix them in the mainline branch first, after which patch backporting is performed to retrofit a patch to stable and LTS branches. However, patch backporting is complex and labor-intensive [6], [54]. It requires maintainers to manually resolve conflicts and adapt patches to out-of-sync branches, even downstream projects, which can be both time-consuming and error-prone.

Recent years have witnessed several proposals on automated patch backporting, which typically consists of two stages: **localization** (figuring out the right location to apply code changes) and **transformation** (adapting code changes to be compatible with the older version).

Localization techniques have evolved throughout the years from exact surrounding-text matching (*e.g.,* as shown in the `patch` utility), to predicates over nodes in typed abstracted syntax tree (AST) (*e.g.,* FIXMORPH [48]), and further to similarities in semantics-bearing program dependency graphs (PDG) (*e.g.,* TSBPORT [70]). While each advancement allows more "fuzzy" contexts to be matched for hosting backported patches, existing solutions still fall short when code in the mainline and target branch undergoes changes beyond what the rules/heuristics are designed for. And such changes can be as simple as renaming a function or adjusting the location of a code snippet (shown in §3) or even declaring a buffer on stack instead of heap (*e.g.,* §2.5)

Similar to localization, patch transformation has also evolved significantly from text drop-in (*e.g.,* `patch`), to typed-AST based transformation rules (*e.g.,* FIXMORPH) and semantics-oriented predefined transformation operators (*e.g.,* TSBPORT). However, existing works still lack the flexibility to adapt or improvise code modification for older branches, especially when the patch transformation does not fall into the predefined templates (in the case of TSBPORT) or rule synthesis search space (in the case of FIXMORPH), such as accurately aligning symbol names (*e.g.,* function call, struct member, header file) between two branches. In addition, existing approaches are tightly coupled with the syntactic and semantic structures of specific programming languages, which makes it impractical to directly apply these tools to backport patches in other languages.

Nevertheless, while evolving localization and transformation techniques draw inspiration from experience of human experts, when human developers backport a patch, they rarely follow an algorithmic procedure to locate patching points nor do they run an exhaustive search on known patterns to adapt the patch. Instead, human developers could perform a mix of activities that may resemble on the lines of: 1) tracing patching location through version control metadata or symbol locator, 2) comprehending patch context changes, and 3) improvising modification to the patch based on the comprehension, and last but not least, 4) trial-and-error. Naturally, one direction of improving backporting tools is to make the tool behave more like human experts.

The recent rise of large-language models (LLMs) shed lights on how we can equip a backporting tool with the capability of code comprehension and code generation. In fact, Mystique [61] and PPatHF [45] has already shown that an off-the-shelf LLM can transform a patch (denoted as function before and after the patch $f_o \rightarrow f_n$) to a new context (f'_o) via in-context learning [5], [75] only given

* The first two authors contributed equally (alphabetical order).

common instructions (I):

$$\text{Query}(I, f_o \rightarrow f_n, f'_o \rightarrow ?), \text{where ? is the LLM response}$$

However, they are not end-to-end backporting systems yet as it still requires developers to manually designate the hosting function (f'_o) for the ported patch. More importantly, they completely forgo rich information (e.g., Git commit history, test cases, etc) that can and will be leveraged in manual backporting. Evidently, they are unable to reflect on or revise incorrect patches.

In this work, we propose PORTGPT, an end-to-end LLM-based patch backporting tool. We intentionally design PORTGPT by shadowing how human developers perform backporting. By atomizing, analyzing, and summarizing key actions taken by developers as well as the information derived from the actions, we note that developers employ a variety of capabilities, such as aggregating Git diffs and counterexample-guided refinement (see §3) in backporting. This implies an agentic architecture—which empowers an LLM more tools and freedom of reasoning—might be more suitable for backporting tasks than in-context learning.

Therefore, the key design philosophy of PORTGPT is to provide tools and chain-of-thoughts that we believe to be useful (based on manual backporting experience) to an LLM agent to facilitate its reasoning. In terms of **localization**, we provide tools for an LLM agent to locate symbols, selectively access relevant source code, and Git history. These tools help the LLM agent focus on the pertinent code and better track code changes across branches. For **transformation**, PORTGPT allows LLM to reason and improvise based on its knowledge base and the concrete backporting task, but also provide best-effort validation of the generated patches and feedback on why a patch cannot be compiled or applied (*e.g.,* missing header file or test case failure).

We built PORTGPT upon GPT-4o [33] and evaluated PORTGPT on large-scale datasets from prior works, including 1465 Linux kernel CVEs from TSBPORT and 350 Linux kernel bugs from FIXMORPH. PORTGPT outperformed both FIXMORPH and TSBPORT with an overall success rate of 89.15%. Additionally, we created a more complex and diverse dataset consisting of 146 cases sourced from 34 programs across three popular languages (C, C++, and Go). On this more challenging dataset, PORTGPT successfully backported 62.33% of the patches, significantly outperforming FIXMORPH and TSBPORT (by 56.53% and 26.09%, respectively).

To evaluate PORTGPT's applicability in real-world, we selected patches for Linux LTS and Ubuntu introduced after the knowledge cutoff (October 2023) of GPT-4o. On the Linux 6.1-stable branch, PORTGPT successfully backported 9 patches out of 18, all of which were thoroughly verified and subsequently accepted by the Linux community. For Ubuntu, we tested 16 patch pairs corresponding to 10 CVEs across multiple versions, and PORTGPT successfully backported 10 of them.

Summary. This work makes the following contribution:

- We propose a practical LLM-based backporting framework that shadows manual patch backporting workflow and leverages commonly used developer tools, such as `git`, to automatically backport patches to target versions.
- We conduct a thorough evaluation of PORTGPT using 1,961 patches, demonstrating that PORTGPT achieves superior performance compared to previous works while maintaining comparable efficiency.
- We leverage PORTGPT to handle real-world patches that are difficult to backport. Nine of these patches are accepted by the Linux kernel community, demonstrating its practical applicability.

We will open-source PORTGPT upon paper acceptance.

2. Why LLMs for Backporting?

Although LLMs have demonstrated exceptional code processing capabilities in a diverse set of scenarios [42], [59], [63], [68], the fundamental reason why LLMs should be considered in backporting is beyond that. In this paper, we argue that the inherent context matching and code transformation schemes in LLMs, while a blackbox in terms of explainability, have the potential to outperform even the most comprehensive set of syntactic or semantic rules or heuristics defined in prior backporting works [48], [70], especially with the right prompting tactics inspired by how human experts solve backporting tasks.

This section presents a series of crafted backporting cases to illustrate how solutions to the two key challenges in backporting—locating matching context and transforming code patch—have evolved from text-based to syntax-based, semantics-based, and finally why LLMs as generative models have a unique advantage over prior rule-based designs.

2.1. Problem Description

Backporting, denoted as $B : P_n \rightarrow P_o$, is a program repair technique that adapts a patch ΔP, originally designed for a newer software version P_n, to an older version P_o. This approach is essential for preserving the security and functionality of legacy systems that cannot be fully updated due to constraints such as compatibility issues or custom configurations. Unlike conventional program repair practices that rely on proof-of-concept (PoC) exploits [32] or static analysis reports [35] to produce the patch ΔP, backporting has access to ΔP already from the beginning. Instead, the core challenges of backporting lies in two aspects:

1) Given ΔP is localized on code context L_n in P_n, we need to accurately locate L_o in P_o that matches L_n and is suitable to host the backported patch.

2) We need to identify a transformation $T(\Delta P, L_o, L_n)$ such that the resulting backported patch integrates seamlessly into L_o, accounting for the differences between L_n and L_o, eliminating the vulnerability addressed by ΔP, and preserving functionalities of P_o.

Stage Setting: suppose we have a simple buffer-overflow vulnerability around code context L_n (Listing 1) and the bug is recently patched with ΔP shown in Listing 2. In the

```c
void process_input(char *input) {
  char buffer[64];
  strcpy(buffer, input);
  printf("Truncated input: %s\n", buffer);
}
```

Listing 1: Code snippet in P_n with a buffer-overflow

```
@@ -1,5 +1,6 @@
 void process_input(char *input) {
   char buffer[64];
-  strcpy(buffer, input);
+  strncpy(buffer, input, 63);
+  buffer[63] = '\0';
   printf("Truncated input: %s\n", buffer);
 }
```

Listing 2: Patch ΔP to fix the bug in P_n (see Listing 1)

rest of this section, we present how backporting tools have evolved in terms of locating L_o and transforming ΔP.

2.2. Vanilla Text-based Backporting

Backporting is almost trivial when there are no (or minimal) changes in the code context where ΔP is applied onto (i.e., L_n). For example, in an older version P_o where the `process_input` function resides in the same file spanning over (almost) the same lines with identical function body shown in Listing 1. In this case, $L_o \approx L_n$ and the transformation of ΔP can be handled by the GNU `patch` utility or `git cherry-pick` gracefully.

2.3. Syntax-based Backporting

The vanilla text-based backporting will stop working both in locating L_o and transforming ΔP when there are even simple syntatic changes, as shown in Listing 3.

In this backporting task, the function was named `handle_input` and the buffer was named `buf` in P_o and somewhere along the development from P_o to P_n, both symbols are renamed (but the vulnerability remains). GNU `patch` errors as it cannot find a matching context for ΔP.

```c
void handle_input(char *input) {
  char buf[64];
  strcpy(buf, input);
  printf("Truncated input: %s\n", buf);
}
```

Listing 3: P_o with the same bug shown in Listing 1.

Syntax-based backporting, piloted by FIXMORPH [48], attempts to solve cases like this by identifying matching contexts and transforming code patches at the level of ASTs. More specifically, locating matching context L_o is essentially checking whether there exists an AST snippet in P_o that matches with L_n, generalized by heuristic rules such as symbol names, type signatures, or even control-flow structures. In the backporting task of Listing 3, FIXMORPH is able to conclude that `handle_input` is the matching context (L_o) for ΔP in Listing 2 due to AST-based fuzzy match (see details in Figure 4 of Appendix). Subsequently, FIXMORPH derives the transformation rule, also expressed on the AST level (shown in Figure 5 of Appendix), and apply the transformation to L_o, creating a backported patch in Listing 4. Note that in the adapted patch, all appearances of `buffer` in ΔP are substituted with `buf` as on both ASTs, var-use sites refer to the actual variable object, not its symbol.

```
@@ -1,5 +1,6 @@
 void handle_input(char *input) {
   char buf[64];
-  strcpy(buf, input);
+  strncpy(buf, input, 63);
+  buf[63] = '\0';
   printf("Truncated input: %s\n", buf);
 }
```

Listing 4: Backported patch for P_o in Listing 3.

2.4. Semantics-based Backporting

While syntax-based backporting is powerful especially when the context matching and patch transformation rules can be expressed with typed ASTs, it can fail when the patch requires semantics (either about the program or the vulnerability it fixes) to backport. Listing 5 is an example when backporting requires the semantic knowledge of the patched vulnerability (buffer overflow). While syntax-based tools (e.g., FIXMORPH) can reconcile symbol changes and produce a patch, the patch will be the same as Listing 4 and this is a wrong patch as now the `buf` is only 32 bytes while both `strncpy(buf,input,63)` and `buf[63]='\0'` in the patch overflow it.

```c
void handle_input(char *input) {
  char buf[32]; // NOTE: length reduced
  strcpy(buf, input);
  printf("Truncated input: %s\n", buf);
}
```

Listing 5: P_o with the same bug shown in Listing 1.

To retrofit semantic information in backported patch, two generic approaches have been proposed in prior works:

Templates. TSBPORT [70] categorizes patches into predefined templates such as adding sanity checks, modifying function call arguments, honoring def-use relations, and use these templates to guide patch transformation. The process involves inferring the intention of each hunk in ΔP, and transforming this hunk to L_o preserving the intention. In the case of Listing 2, TSBPORT infers the intention of the only hunk in ΔP is to use safer string functions and null-terminate a buffer, and subsequently transforms the patch to Listing 6 which applies to Listing 5 correctly.

Constraint-solving. PATCHWEAVE [49] retrofit semantics into backporting via concolic execution. More specifically, it summarizes the semantic effect of ΔP (preventing a buffer overflow) and ensure that the backported patch achieves the same effect. However, PATCHWEAVE requires a PoC input in order to collect symbolic constraints around the patch, which is not always readily available. And symbolic execution inherently struggle to scale for large codebases.

```
@@ -1,5 +1,6 @@
 void handle_input(char *input) {
   char buf[32]; // NOTE: length changed
-  strcpy(buf, input);
+  strncpy(buf, input, 31);
+  buf[31] = '\0';
   printf("Truncated input: %s\n", buf);
 }
```

Listing 6: Backported patch for P_o in Listing 5.

2.5. LLM-based Backporting

While effective, semantics-based backporting, especially those relying on heuristically defined templates, can still fail when the tool cannot infer the intention of the patch (or a hunk in the patch), or cannot transform a hunk as the changes in the context (L_o) is not captured by any of its templates. Listing 7, it is only a small tweak to Listing 5 by allocating the `buf` on heap instead of stack, but TSBPORT cannot produce a correct patch as inferring the size of `malloc`-ed object is not a template in TSBPORT. However, backporting ΔP via any modern LLM (e.g., ChatGPT 4o) is simple even with the most obvious prompt, as shown in the shared conversation. Based on this experience, we believe LLM-based backporting can be a catch-all solution, especially for changes beyond what is describable by heuristics and rules in syntax- and semantics-based backporting practices. This stance, to the best of our knowledge, is not highlighted in prior works yet.

```
void handle_input(char *input) {
  char *buf = malloc(32);
  if (buf == NULL) { return; }
  strcpy(buf, input);
  printf("Truncated input: %s\n", buf);
  free(buf);
}
```

Listing 7: P_o with the same bug shown in Listing 1.

2.6. Backporting Types

In this work, we adopt the same categorization for backporting tasks established in prior studies [48], [70]. Based on how significantly it differs from the original patch, a backported patch is typically classified as:

- **Type-I** *(No changes)*: The backported patch is identical to the original patch both in code and location, meaning it can be cherry-picked trivially without any modifications.
- **Type-II** *(Only location changes)*: The backported patch requires no transformation, it differs from the original patch only in where it is applied (*e.g.,* line numbers or file names).
- **Type-III** *(Syntatic changes)*: The backported patch requires syntatic modifications only (*e.g.,* function and variable names) to ensure compatibility with the target version.
- **Type-IV** *(Logical and structural changes)*: More intrusive modifications are applied (*e.g.,* adding or removing lines in the patch), making the backported patch syntactically different while preserving the same functionality.

3. Prompting (In-context Learning) Is Not All

While §2 highlights the inherent advantages of LLMs over traditional methods for patch backporting, a critical question remains: how can we best leverage and enhance their code understanding capabilities to navigate the complexities of diverse backporting scenarios? Indeed, a natural approach is to enable LLMs to emulate the processes employed by human developers during backporting. In this section, we present an example illustrating how human developers perform patch backporting. This demonstrates how we should equip LLM with a similar set of capabilities to emulate human developers effectively.

3.1. Motivating Example

CVE-2022-32250. Listing 8 presents the patch in the original version for a use-after-free (UAF) bug that caused CVE-2022-32250 [12]. To address this issue, the core logic of the patch involves relocating the check statements from `nft_set_elem_expr_alloc` to an earlier stage in the process, specifically within `nft_expr_init`. However, this patch cannot be directly applied to *5.4-stable*, and a call-for-volunteer is even advertised on the mailing list [13].

Backport by human. Human experts typically start with the `patch` utility first, and then handle failed hunks on a hunk-by-hunk basis. Unfortunately, in this example, neither of the two hunks can be applied directly due to context conflict, *i.e.,* the code context that the patch involves has differed between the original version and the target version.

❶ To **locate** the first hunk in the target version, the expert begins by finding the definition of function `nft_expr_init`. This localization is identified by its semantic similarity to the patch; in this instance, a key `goto` statement, unique to the patch's logic, pinpoints the location. ❷ Observing that the patch's surrounding context had evolved between versions, to **transform** the hunk, the expert then consulted git history to understand the reasons for these changes before attempting conflict resolution. ❸ Subsequently, the expert examines the change history of the code snippet through `git` and resolves the namespace conflicts that `expr_info` should be replaced with `info` in the target version. ❹ with that, the expert completes the first hunk of the patch.

❺ To **locate** the second hunk in the target version, the expert first attempts to find function `nft_set_elem_expr_alloc` by its symbol but fails as the function name does not exist in the old version. ❻ Then, the expert traces the origin of this function in the original version with `git` and finds that it was introduced by commit *a7fc93680408*. ❼ Next, the expert uses `git show` to examine the details of this commit and concludes that `nft_set_elem_expr_alloc` was migrated from `nft_dynset_init` in another file. ❽ Therefore, the expert views function `nft_dynset_init` in the old version instead and makes sure it is the correct host for backported patch. ❾ Finally, referencing the git history, the expert infers that `priv->expr` in the target version is

```
--- a/net/netfilter/nf_tables_api.c
+++ b/net/netfilter/nf_tables_api.c
@@ -2873,27 +2873,31 @@ *nft_expr_init(
  err = nf_tables_expr_parse(ctx, nla, ...);
  if (err < 0) goto err1;

+ err = -EOPNOTSUPP;
+ if (!(expr_info.ops...flags & NFT_EXPR_STATE))
+   goto err_expr_stateful;

  err = -ENOMEM;
  expr = kzalloc(expr_info.ops->size, ...);
@@ -5413,9 +5417,6 @@ *nft_set_elem_expr_alloc(
    return expr;

  err = -EOPNOTSUPP;
- if(!(expr->ops->type->flags & NFT_EXPR_STATE))
-   goto err_set_elem_expr;
-
  if (expr->ops->type->flags & NFT_EXPR_GC) {
    if (set->flags & NFT_SET_TIMEOUT)
      goto err_set_elem_expr;
```

Listing 8: Original Patch for CVE-2022-32250

equivalent to `expr` in the patch, and consequently deletes the corresponding statements.

❿ After completing the patch backporting for the target version, the expert runs tests to verify its correctness (as in no regression) and effectiveness (as in defending against the PoC exploit, if available). Additionally, the expert refines the patch by addressing potential issues identified during compilation, testsuite evaluations, and PoC testing.

3.2. Observations

Based on the manual patch backporting process presented above, we draw some key observations that influence the design of PORTGPT—our LLM-based backporting tool.

1) It is impossible to fit all information possibly needed by end-to-end backporting (*e.g.,* the entire commit history) into a single or even a pre-defined sequence of prompts.

2) Information relevant to the current step can be retrieved frequently on an as-needed basis, but it still needs to be presented in a concise way in order not to overload the context limit of a human brain.

3) Decisions are frequently taken based on results of previous steps in a trial-and-error manner, *e.g.,* tracing the provenance of a symbol when symbol resolution fails while viewing function body of lookup is successful.

4) Backported patch doesn't have to be generated in one-take. Validating and subsequently improving the patch based on feedback is an inseparable step in the manual process.

Based these observations, we believe an agentic design is well-suited for LLM-based backporting, and identify three key capabilities to retrofit a certain degree of human expertise, common workflows, and best-practices into the agent.

Basic code viewing and symbol lookup: It is essential to equip LLM with basic code access capabilities, as what human expert does in steps ❶❷❽❽. By utilizing symbol locating and code viewing, LLM can access the code context of the target version.

Access aggregated diff and track provenance : By aggregating historical commits of the patch code, we can precisely trace the location of the code change in the target version, as shown in steps ❸❻❼. Whether it involves code relocation or identifying code segments absent in the target version, analyzing the modification history allows for precise determination. Moreover, the history of the changes clearly reveals the changes of the namespace present in the context.

Counterexample-guided refinement: Since the patch from the original version may include dependencies (*e.g.,* header file imports) unavailable in the old version, basic compilation tests are essential. If there are compilation errors, the previously generated patch can serve as a counterexample, using compiler error messages to guide the refinement of the patch. This step is also a practice typically performed by experts after developing the patch, as shown in step ❿.

4. Design Details of PORTGPT

In order to empower LLM agents with the capabilities presented in §3, we design a series of tools and workflows and further implement several optimizations to standardize and improve the formatting of LLM outputs. In this section, we describe these design details of PORTGPT. We start with an overview of PORTGPT (§4.1), followed by detailed description of two main stages of PORTGPT: Per-Hunk Adaptation (§4.2) and Final Patch Combination (§4.3).

4.1. Overview

As illustrated in Figure 1, the overall workflow of PORTGPT consists of two main stages: Per-Hunk Adaptation and Final Patch Combination. In the first stage, PORTGPT completes **localization** and initial **transformation** for each hunk individually. PORTGPT extracts hunks from the original patch and per each hunk, PORTGPT first determines whether the hunk requires backporting, and if so, transforms the hunk to ensure compatibility with the target version. This stage aims to ensure that the generated patch can be successfully applied to the target version. In the second stage, PORTGPT combines the transformed hunks, applies the entire patch to the target version, and sends the backported codebase for compilation. If compilation failed, PORTGPT attempts to resolve them by adding necessary definitions or adjusting the code context to finalize the **transformation**. Ultimately, PORTGPT outputs backported patches that are free from compilation errors. Both stages leverage an LLM agent, supplemented with customized tools designed to enhance the LLM's performance.

4.2. Per-Hunk Adaptation

Backporting a hunk to the target version takes two steps. First, it is necessary to determine if the hunk should be backported and to identify the appropriate matching context. Second, the hunk must be transformed to match the host context in the target version. To complete these

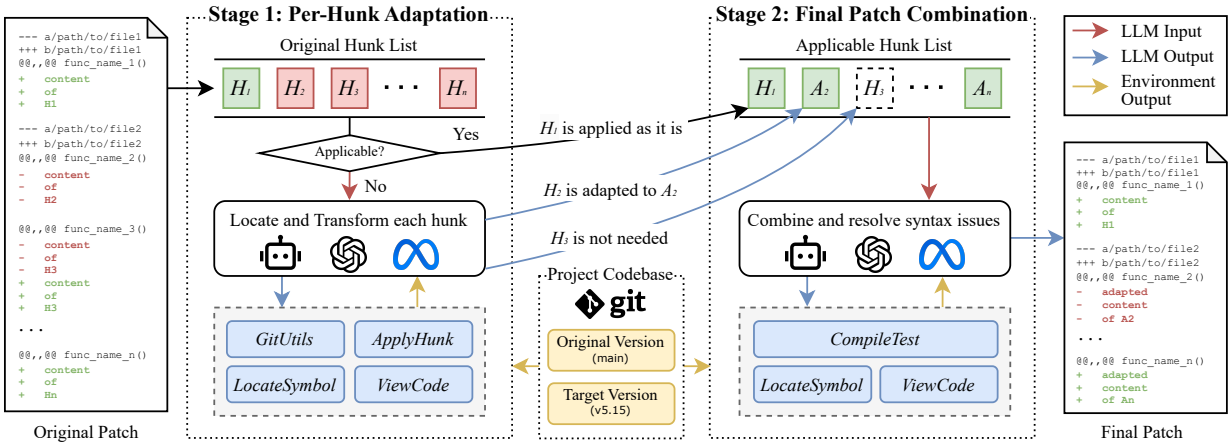

Figure 1: **Workflow of PortGPT**

steps, PortGPT must gather sufficient information from the codebase to understand how the codebase has evolved across different branches and how the current hunk relates to the surrounding code. To support this process, we have designed three tools called *ViewCode*, *LocateSymbol*, and *GitUtils* to facilitate information retrieval. Once a hunk is transformed, PortGPT uses another tool called *ApplyHunk* to apply the adapted hunks to the target version.

• ***ViewCode*.** This tool takes four parameters: *ref*, *file*, *start*, and *end*, which specify the commit, file path, and the line range to be retrieved. It returns the code snippet between the specified line numbers in the given version of the codebase, enabling the LLM to access any portion of the source code from any Git-tracked version.

• ***LocateSymbol*.** This tool takes two parameters: *ref* (the specific commit) and *symbol* (language objects, such as function name, variable name). It returns the definition site of the symbol in the specified version, including the file path and line number, allowing the LLM to accurately locate symbols within the codebase.

• ***GitUtils*.** This tool consists of two components: *History* and *Trace*, neither of them directly takes any parameters from the LLM. The *History* component presents the evolution of code snippets across commits or versions, while *Trace* provides detailed insights into the trace of code movement. They are designed to provide information on code evolution and commit history to the LLM. The detailed design will be discussed later.

• ***ApplyHunk*.** This tool accepts one parameter: *patch* and allows LLM to apply the patch to the target version. To mitigate potential errors in LLM-generated patches, it automatically corrects patch formatting issues and generates diagnostic feedback for the LLM if the patch failed to apply. Further details are provided below.

***GitUtils* Detail.** The *History* component displays a list of Git commits that modify the code snippet within the current hunk from the *fork point* to the original version. The *fork point* refers to the divergence between the original and target versions, *i.e.,* the most recent common ancestor. For example, v6.1 tag is *fork point* of the mainline branch and the linux-6.1.y stable branch. The tool analyzes the file modification history and identifies all changes made from the *fork point* to the original version. It further filters the history to include only changes affecting the lines within the hunk. This design ensures that the output is both comprehensive and relevant. By focusing on commits from the *fork point* onward and isolating the changes to the lines in the hunk, *History* provides a detailed evolution of the code while excluding unrelated modifications.

While the component *History* reveals local changes in the corresponding code snippet of a hunk, it may overlook critical global changes, such as: the relocation of code. These global changes, though important, are challenging to incorporate directly into *History* because they are often too lengthy for an LLM.

To address this limitation, we introduce *Trace* to extract and highlight significant global changes, particularly the relocation of patch contexts. We observe that code movement changes have a direct impact on backporting. Code movement refers to the relocation of a code snippet from one location to another after the *fork point*, resulting in discrepancies in its placement between the original and target versions. To produce an accurate backported patch, LLM must identify the code movement and its target location in a patch. Leveraging the minimal edit distance method [52], we compare the added code with the deleted code to facilitate LLM in the code movement identification. Edit distance is an effective metric—it captures syntactic similarity and is applicable across different programming languages. Finally, with the location of the matched code and its associated commit message provided by *Trace*, LLM determines if the code is migrated from that location or the code is newly introduced.

***ApplyHunk* Detail.** This tool systematically handles errors that may arise during `git apply`. For each type of error, we either follow a customized correction scheme or generate corresponding feedback to guide the LLM for an iterative refinement of the patch (*i.e.,* the transformed hunk).

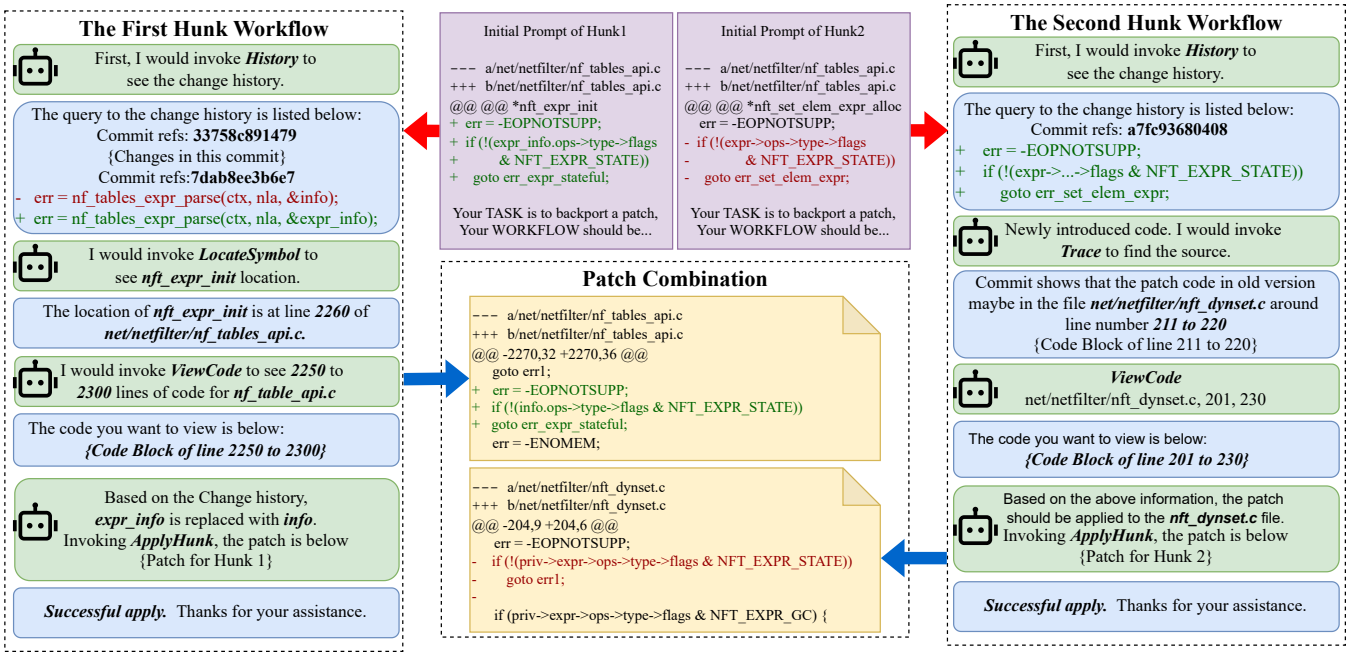

Figure 2: **Example of Per-Hunk Adaptation.** The red arrows represent LLM agent inputs, while the blue represent LLM agent outputs. The green box represents the invocations of LLM, while the blue box represents the outputs of tools.

Corrupted patch. This error arises when the patch does not adhere to the correct format [28], *i.e.,* LLM forgets to add a space at the beginning of the code line. Therefore, the tool avoid such error by checking each code line of the patch and adding required spaces.

Non-existent file path. This error occurs when the patch is applied to a file that does not exist, often due to the file being renamed or the relevant function residing in a different file in the original version. To address this, the tool tries to identify possible file path lists. Initially, the tool checks if the file has been renamed. If not, the tool locates possible files in the target version by either matching the original version file name or locating one symbol referenced in the patch. Finally, the tool applies the patch to all identified possible file paths and returns apply feedback to LLM.

Context mismatch. Patch context refers to lines that start with ' ' (space) or '-' in the patch. This error occurs when the context of the LLM-generated patch (C_l) is inconsistent with the target code. To resolve this, this tool identifies the most similar code block with the minimal edit distance in the target version as C_t. Then, it compares C_l with C_t line by line and highlights any discrepancies for the LLM, allowing it to make the necessary adjustments. Furthermore, we propose an **Automated Context Correction** mechanism, which forcibly substitutes lines in C_l that differ from C_t before applying the patch. Once the correction is made, the patch can be successfully applied. To preserve the LLM's original intent as much as possible, the tool only activates automated context correction after three failed attempts to apply the patch due to context mismatches.

Patch Localization. Locating a matching context in the target version to host an adapted hunk includes two subtasks: identifying the correct target file path and locating the corresponding code block within that file. We find possible locations by applying the original hunk with *ApplyHunk* and provide possible locations in tool's output to LLM. LLM could adopt two distinct approaches for using *GitUtils* to narrow down to the correct location.

Symbol-Based Localization: This workflow is particularly effective when the patch does not involve newly introduced code or complex code movement. In this approach, the LLM first invokes *History* to retrieve the code change history and determine whether the original symbol name has been renamed in the target version. If renamed, it then uses *LocateSymbol* to identify the location of the renamed symbol within the target codebase. Cooperated with *ViewCode*, LLM further checks code blocks around those symbols and determine where the patch should be backported to.

Movement-Aware Localization: In more complex scenarios, such as a code snippet has been moved to several different locations, *LocateSymbol* may not be sufficient to identify the required symbols. In such cases, the LLM invokes *Trace* to determine the specific location of the code in the target version or to confirm if the code no longer exists. This ensures accurate localization and facilitates modifications even in challenging situations.

CVE-2022-32250. To better understand the operations in the first stage, we continue with the patch of CVE-2022-32250 (shown in Listing 8). The overview of this process is provided in Figure 2.

Regarding the first hunk, the LLM sequentially performs the following steps: ❶ Invokes *GitUtils*. From *History*, the LLM observes, `expr_info` has been replaced with `info` in the old version. ❷ Invokes *LocateSymbol*

Algorithm 1: How *ApplyHunk* corrects a patch and generates feedback for different failure reasons.

Input: A patch to be corrected and applied; A flag indicates whether to activate the Automated Context Correction mechanism.
Output: Patch apply feedback;

```
Function apply_feedback(patch, flag):
    output = str();
    possible_files = list();
    patch = format_patch(patch); // avoid "invalid patch"
    if flag then
        patch = automated_context_correction(patch);
    result = git_apply(patch);
    if "File not found" in result then
        old_file = find_rename_file(patch);
        if old_file then
            return old_file;
        possible_files.append(locate_symbol(patch));
        possible_files.append(find_similar_file(patch));
        for path in possible_files do
            patch = revise_patch(patch, path);
            feedback = apply_feedback(patch);
            if "success" in feedback then
                return path;
            output = output + path + feedback;
        return output;
    if "Error while searching for context" in result then
        C_llm = extract_context(patch);
        C_tgt = find_corresponding_context(C_llm);
        return compare(C_llm, C_tgt);
    return "success";
```

to determine the location of `nft_expr_init` (modified by the hunk) in the old version. ❸ Invokes *ViewCode* to inspect the code context surrounding the identified location. ❹ Generates the transformed patch based on the gathered information and then invokes *ApplyHunk* to validate its correctness. For the second hunk, it involves more complex scenarios. The LLM first ❶ invokes *History*, which shows that the current code context was introduced in the *a7fc93680408* commit. This satisfies the conditions for calling *Trace*. Consequently, the LLM proceeds to ❷ invoke *Trace*. Through minimal edit distance similarity matching, *Trace* identifies that this code originates from Line 211 to 220 in `net/netfilter/nft_dynset.c`. With precise code location information, the LLM then follows the same workflow and invokes ❸ *ViewCode* and ❹ *ApplyHunk* to complete the patch transformation. After transforming both hunks, PORTGPT generates the combined patch.

```
1  --- a/fs/cifs/cifs_debug.c
2  +++ b/fs/cifs/cifs_debug.c
3  @@ -332,6 +332,11 @@
4   struct cifs_ses *ses;
5   list_for_each(...) {
6  +    spin_lock(&ses->ses_lock);
7  +        if (ses->ses_status == CifsExiting) {
8  +                spin_unlock(&ses->ses_lock);
9  +                continue;
10 +        }
11          if ((ses->serverDomain == NULL) ||
```

(a) Patch Generated by Stage-1

```
1  --- a/fs/cifs/cifs_debug.c
2  +++ b/fs/cifs/cifs_debug.c
3  @@ -332,6 +332,11 @@
4   struct cifs_ses *ses;
5   list_for_each(...) {
6  +    spin_lock(&GlobalMid_Lock);
7  +        if (ses->status == CifsExiting) {
8  +                spin_unlock(&GlobalMid_Lock);
9  +                continue;
10 +        }
11          if ((ses->serverDomain == NULL) ||
```

(b) Patch Generated by Stage-2

Figure 3: Patches Generated for CVE-2023-52752.

4.3. Final Patch Combination

After the inital patch is generated from previous stage, PORTGPT will combine all the generated patches of each hunk and check for compilation errors. If there are any errors, PORTGPT will resolve them by adding necessary definitions or adjusting the code context, which is achieved by an LLM equipped with a series of customized tools which included *ViewCode* and *LocateSymbol* we mentioned before with a new tool *CompileTest*. Same as tool *ApplyHunk*, *CompileTest* takes a draft patch as input, the tool will apply the patch to the target version and compile the code. If the code can be compiled successfully, the tool will output the final patch and terminate the second stage. Otherwise, the tool will return the error message. Typically, the compilation error messages are long and complex, which may degrade the performance of the LLM [40]. To address this issue, we design a customized error message parser to extract the key information from the error message. The parser is based on a set of regular expressions that are designed to capture the most common error patterns. The parser extracts the error type, the file name, the line number, and the error message. The extracted information is then used to guide the LLM in resolving the compilation errors.

To better understand how the second stage works, we provide an example. Figure 3a presents the patch for CVE-2023-52752 [18] generated by Stage-1. However, the `cifs_ses` structure (Line 4) does not have members `ses_lock` and `ses_status`, leading to a compilation error. PORTGPT will issue an error message to the LLM: "`fs/cifs/cifs_debug.c` at Line 340, `cifs_ses` has no member `ses_lock`" (`ses_status` same), requesting a fix. Based on this information, the LLM will follow these steps: ❶ First, it invokes *LocateSymbol* to find the

definition of `cifs_ses`, with *LocateSymbol* outputting that the symbol is defined in `fs/cifs/cifsglob.h` at Line 968. ❷ Then, it invokes *ViewCode* to inspect the definition of `cifs_ses`. In the definition, `ses_status` corresponds to a similar field `status`. The comment also indicates that `status` is protected by `GlobalMid_Lock`. ❸ Based on the content from *ViewCode*, the LLM deduces that `ses_status` in the target version should be replaced by `status`. The semantics indicate that `ses_status` is protected by `ses_lock`, so `status` should be protected by `GlobalMid_Lock` accordingly. The LLM will then generate the revised patch in Figure 3b and invoke the validation tools to verify its correctness.

4.4. Prompt Design

To enable the LLM to perform the backporting task more effectively, we carefully craft the prompt for each stage:

Agents in the two stages share the same system prompt, which explains the concept of patch backporting, the provided tools and their usage. This ensures that the LLM understands the context of patch backporting and the resources it can access. The user prompts differ between two stages since they are different tasks (shown in Figure 6 in Appendix). Notably, both user prompts consist of two key components: the task, which specifies what the LLM needs to accomplish, and the workflow, which provides detailed guidance on how to execute the task effectively.

In the first stage, the objective is to generate a patch that can be successfully applied hunk by hunk. The user prompt specifies the patch to be backported, the original version, and he target version and provides a clear workflow to guide the LLM. Specifically, LLM firstly locates where to apply the code changes in the target version by: ❶ utilizing similar code blocks we provided in the prompt, ❷ invoking *GitUtils* to to analyze code change history, ❸ invoking *LocateSymbol* to identify the locations of functions or variables. The user prompt also instructs the LLM to transform a hunk to align with the target version's context, leveraging tools such as *ViewCode* to access and analyze the target version's code.

In the second stage, the objective is to ensure that the complete patch is reliable and does not introduce new risks. The LLM is prompted to iteratively refine the patch based on validation feedback, which may include compilation error messages produced by the *CompileTest* tool.

In summary, the prompts are structured to empower an LLM for backporting tasks by ensuring it knows how to use the tools to gather as much necessary information as possible while also following a clear and systematic workflow.

5. Implementation

PORTGPT uses LangChain [39] to facilitate the integration of LLM agents and prompt engineering, and existing tools for symbol resolution and patch validation.

Locate Symbol. For symbol resolution, we use **ctags** [57] as the syntax parser. As a tool widely used in text editors for fast navigation, ctags supports syntax parsing for dozens of programming languages, such as C, C++, Go. In PORTGPT, by parsing the output of ctags, we could quickly generate an available symbol table for *LocateSymbol*. Specifically, ctags parses symbols such as functions, structs, and variables that appear in the codebase. It then outputs the symbol's location information in the format of "`symbol_name file lineno symbol_type`". By reading the output in the ctags format, we can generate a symbol table with symbols and their corresponding locations.

Validation Chain. To ensure that the LLM-generated patch does not introduce new security risks when both the PoC and functionality test suites are available, we design a rigorous, multi-stage validation chain following the second stage. First, we execute the original PoC on the patched version to confirm that the vulnerability has been successfully eliminated. And then, we run comprehensive functionality test suites to verify that the patch preserves the intended behavior and does not compromise the program's correctness. Finally, we would additionally validate a backported patch if a directed fuzzer suitable for the underlying project can be used. This layered validation strategy significantly enhances the reliability and safety of the generated patches.

6. Evaluation

In this section, we assess PORTGPT to address the following research questions:

- **RQ1 (Performance)**: How does PORTGPT compare to other works in terms of the success rate of backporting?
- **RQ2 (Ablation)**: How does each module in PORTGPT contribute to its overall performance?
- **RQ3 (Efficiency)**: How well PORTGPT handles the backporting task in terms of time and cost efficiency?
- **RQ4 (Practicality)**: How applicable is PORTGPT for handling backporting tasks in real-world scenarios?

6.1. Settings

Environment. All experiments were conducted on a 64-bit Ubuntu 24.04 LTS server equipped with an Intel(R) Xeon(R) Gold 6248R 96-core CPU running at 3.00 GHz, 512 GB of memory, and a 22 TB hard drive. The large language models used in our evaluation include GPT-4o [33] (version *gpt-4o-2024-08-06*), Llama 3 [36] (version *Llama 3.3 70B*) and DeepSeek-v3 [27] (version *DeepSeek-V3-0324*).

Datasets. We evaluate PORTGPT using 1961 patches, which include 350 patches from FIXMORPH [48], 1465 patches from TSBPORT [70]. And 146 patches collected by ourselves (details in Table 5 in Appendix). These patches span 34 programs across three languages (C, C++, and Go), with the number of modified lines ranging from 1 to 1792. These 146 patches are collected based on the following criteria: ❶ The corresponding vulnerability of the patch is recorded in the CVE database within the past four years. ❷ At least one version within the same project has not reached end-of-life and requires backporting. ❸ The patch cannot be trivially applied to the target version, *i.e.,* not Type-I.

Dataset	System	Type-I	Type-II	Type-III	Type-IV	Total
Prior works [48], [70]	FIXMORPH	20/170 (11.67%)	374/1208 (30.96%)	23/92 (25.00%)	30/345 (8.70%)	447/1815 (24.63%)
	ChatGPT†	67/170 (39.41%)	451/1208 (37.30%)	16/92 (17.58%)	22/345 (6.38%)	556/1815 (30.63%)
	TSBPORT	170/170 (100.00%)	1190/1208 (98.51%)	69/92 (75.00%)	160/345 (46.38%)	1589/1815 (87.59%)
	TSBPORT*	162/170 (95.29%)	919/1208 (76.08%)	61/92 (66.30%)	150/345 (43.48%)	1292/1815 (71.18%)
	PORTGPT	170/170 (100.00%)	1186/1208 (98.18%)	74/92 (80.43%)	188/345 (54.49%)	1618/1815 (89.15%)
Ours (C)	FIXMORPH	N/A	0/6 (0.00%)	3/29 (10.34%)	0/34 (0.00%)	3/69 (4.34%)
	TSBPORT	N/A	5/6 (83.33%)	14/29 (48.28%)	5/34 (14.71%)	24/69 (34.78%)
	TSBPORT*	N/A	2/6 (33.33%)	14/29 (48.28%)	4/34 (11.76%)	20/69 (28.99%)
	PORTGPT	N/A	6/6 (100%)	21/29 (72.41%)	15/34 (44.12%)	42/69 (60.87%)
Ours (C++)	PORTGPT	N/A	N/A	10/11 (90.91%)	5/17 (29.41%)	15/28 (53.57%)
Ours (Go)	PORTGPT	N/A	13/13 (100%)	7/9 (77.78%)	14/27 (51.85%)	34/49 (69.39%)

TABLE 1: **Performance Comparison of Backporting.** The first two sections of the table compare PORTGPT with FIXMORPH and TSBPORT using datasets from prior works as well as C cases on our own dataset. The last two rows display the performance of PORTGPT on C++ and Go cases from our dataset. ChatGPT† is from TSBPORT [70]. **TSBPORT*** represents the performance of TSBPORT under the same settings as PORTGPT and FIXMORPH. Unlike PORTGPT and FIXMORPH, TSBPORT requires the target file for each hunk as input, which may not be available in real-world scenarios. To ensure a fair comparison, we introduce **TSBPORT*** as a variant of TSBPORT that operates without access to the target file, simulating a more realistic environment.

Evaluation Criteria. We compare PORTGPT with FIXMORPH and TSBPORT using their datasets (1,815 cases in total) along with all C cases in our dataset. Comparisons with C++ and Go cases from our dataset were excluded, as both tools support C only. Although we attempted to adapt these tools for use with C++ and Go, their implementations are highly tailored to C, making migration infeasible.

Additionally, unlike PORTGPT and FIXMORPH, TSBPORT requires the target file for each hunk as input—a condition that assumes some degree of ground-truth patch localization, which is not entirely realistic. To ensure a fair comparison, we introduce a variant of TSBPORT, denoted as TSBPORT*, that operates under the same settings as PORTGPT and FIXMORPH. TSBPORT* is designed to function without access to the target file, thereby simulating a more realistic environment. By aligning the evaluation conditions, we enable a more practical and meaningful comparison of the performance among the three tools.

We excluded SKYPORT [51] from our comparison, as it is designed for web applications (PHP only), which is not included in our dataset. Mystique [61] and PPATHF [45] was excluded as it is designed for function-level patch porting and requires perfect localization, whereas backporting typically involves multiple functions. Although we attempted to compare PATCHWEAVE [49], its dependence on KLEE symbolic execution presents challenges. The use of symbolic execution can result in path explosion, which makes it impractical for large-scale programs (*e.g.,* Linux kernel). Furthermore, PATCHWEAVE requires PoC for the corresponding vulnerabilities, which is not available in our dataset.

Validation Methodology. To validate the correctness of the patches generated by PORTGPT, we compare them with the ground truth. If the patches are not identical, we conduct human validation to ensure their correctness, following best practices outlined in [48], [70]. Each patch is independently reviewed by three researchers to guarantee accuracy and consistency. We also try our best to collect PoC and test suites for each vulnerability in our dataset, following the validation chain described in §5. To evaluate whether PORTGPT-generated patches introduce new security vulnerabilities, we run directed fuzzing (SyzDirect [55]) on successfully backported Linux patches that differ from ground-truth. No security risks were identified during 4 hours of fuzzing.

6.2. Performance Evaluation (RQ1)

Table 1 provides a summary of PORTGPT's performance, which is based on GPT-4o, in comparison to FIXMORPH and TSBPORT, while Table 2 illustrates PORTGPT's performance when powered by different LLMs.

6.2.1. Comparison with FIXMORPH. In the dataset from prior works [48], [70], PORTGPT achieves notable performance enhancements across all backporting types when compared to FIXMORPH. For instance, in Type-IV patches, FIXMORPH demonstrates a modest success rate of 8.70% while PORTGPT improves it to 54.49%, showcasing its resilience in addressing complex backporting scenarios. The performance advantage of PORTGPT is more remarkable in our dataset for C test cases. Across all types, FIXMORPH achieves a success rate of 4.34%, PORTGPT demonstrates superior efficacy with a success rate of 60.87%.

We analyzed the failure cases of FIXMORPH within our dataset. Firstly, FIXMORPH enforces stricter constraints by supporting only C source files while excluding C header files. This limitation is explicitly stated in the paper and is further confirmed by its implementation, which resulted in 14 failed cases. Among other failed cases, FIXMORPH failed to produce results in 33 instances, primarily because it raised errors that caused the transformation process to terminate

prematurely. In seven of these failures, FIXMORPH was unable to identify the correct patch locations in the target version. In the remaining cases, its AST-based approach was unable to handle complex changes, such as code syntax restructuring and module-level semantic modification, resulting in failures during the AST transformation process.

FIXMORPH generated a total of 22 patches out of which 19 are incorrect, and they exhibit several issues. These patches contained syntax errors, such as disordered statements or failure to utilize functions or variables compatible with target versions. Furthermore, FIXMORPH struggled to accurately migrate patches involving changes in semantics. Moreover, the complexity of the code plays a crucial role in the performance of FIXMORPH. The average number of modified lines for successfully backported patches is 1.3, compared to 40.1 for failed patches. This difference indicates that FIXMORPH struggles with cases involving intricate logic and extensive code modifications.

6.2.2. Comparison with TSBPORT. In the dataset from prior works [48], [70], PORTGPT achieves comparable results to TSBPORT on simpler patch types but excels in handling more complex types, with success rates of 80.43% and 54.49% on Type-III and Type-IV, respectively, compared to 75.00% and 46.38% in TSBPORT. On our dataset, the advantage of PORTGPT becomes even more evident. It achieves 100% accuracy on Type-II cases and significantly outperforms TSBPORT on Type-III (72.41% vs. 48.28%) and Type-IV (44.12% vs. 14.71%).

In addition, TSBPORT requires manual designation of the target file for each hunk and we provide it with the ground truth in the experiments, which favors TSBPORT. In particular, TSBPORT does not handle the issue of mismatched target and original patch files, which is the main challenge in Type-II (as these patches requires no transformation at all). To address this evaluation bias, we limit files from the original patch as the target in TSBPORT*. PORTGPT outperforms TSBPORT* by 17% overall.

The underperformance of TSBPORT on Type-III and IV cases in our dataset stems from several issues. For Type-III, which involves 15 failure cases, the primary reason is erroneous symbol importing. TSBPORT often struggles to resolve missing symbols, such as function or global variable declarations in header files, during patch migration. This limitation frequently leads to failed patch applications or introduction of additional header files imports, accounting for errors in 11 out of the 15 cases. These extraneous header file imports are typically migrated from the header files in the original version, but these headers do not exist in the target version, causing compilation error.

Type-IV encompasses 29 failure cases, which can be attributed to two primary issues. The first issue is TSBPORT's inadequate handling of module-level modifications, such as updates to struct fields or the introduction of new functions. This deficiency leads to incorrect mappings, accounting for 7 out of the 29 failures. The second issue is the complexity of the code itself. This is evidenced by a notable difference in the average number of modified lines: 13.8 for successful patches versus 35.7 for failed ones. The contrast highlights TSBPORT's difficulty in dealing with complex logic and extensive modifications. Note that PORTGPT faces the same difficulty as well—more complicated patches are harder to backport—but with a higher bar since the average number of modified lines for successful patches reached 20.1, while for failed patches, it averaged 51.0.

Model	Type-II	Type-III	Type-IV	Total
GPT-5	19/19	42/49	31/78	92/146
GPT-4o	19/19	38/49	34/78	91/146
Gemini-2.5-Flash	19/19	37/49	27/78	82/146
DeepSeek-v3	18/19	33/49	25/78	77/146
Llama-3.3	17/19	7/49	0/78	24/146

TABLE 2: **PORTGPT Performance based on Different Large Language Model.**

6.2.3. Generalizability over other programming languages. PORTGPT demonstrates strong programming language generalizability. Unlike FIXMORPH and TSBPORT, which are coupled with C only, PORTGPT consistently performs well across multiple programming languages. Notably, adapting PORTGPT from C to C++ and Go required no modification, highlighting its flexibility. These results underscore PORTGPT's potential for robust generalizability and its capability to be effectively extended to support a broader range of programming languages. TSBPORT and FIXMORPH required predefined transformation rules based on expert knowledge, whether for syntax or semantic matching, and were tightly coupled with specific programming languages. In contrast, PORTGPT relies on a series of text-based tools (e.g., GitUtils) for localization and leverages the code generation capabilities of LLM to perform patch transformation. This allows PORTGPT to be language-agnostic.

However, PORTGPT's performance on Type-IV C++ cases is significantly lower, with a success rate of only 5 out of 17 cases (29%) compared to at least 40% in other languages. This discrepancy may partly stem from the small dataset, which could introduce statistical bias. Another contributing factor is the complexity of required modifications: 11 out of 12 C++ failure cases necessitate around 100 lines of changes, whereas most C cases require fewer than 50 lines.

6.2.4. Effectiveness of other large language models. As shown in Table 2, PORTGPT demonstrates varying performance across different language models. GPT-5 achieves the best overall performance with 92/146 (63.0%) success rate, followed closely by GPT-4o with 91/146 (62.3%). Gemini-2.5-Flash delivers solid performance with 82/146 (56.2%) overall success, while DeepSeek-v3 shows moderate effectiveness at 77/146 (52.7%). This decline in Llama 3.3 performance is primarily attributed to Llama 3.3's limited function-calling capabilities, which are critical for retrieving information in the backporting task. Notably, Llama 3.3 often aborts processes prematurely when faced with insufficient information, instead of attempting additional

Configuration	Type-II	Type-III	Type-IV	Total
w/o GitUtils	94.74%	75.51%	32.05%	54.79%
w/o Locate	94.74%	69.39%	38.46%	56.16%
w/o Loc & Git	84.21%	59.18%	25.64%	44.52%
w/o AutoFix	100%	65.31%	30.77%	51.37%
w/o Compile	100%	69.39%	25.64%	50.00%
PORTGPT	**100%**	**77.55%**	**43.59%**	**62.33%**

TABLE 3: **Ablation Study of PORTGPT.** w/o = without. GitUtils, Locate (LocateSymbol) and Compile (CompileTest) are three tools in PORTGPT. AutoFix refers to the patch correction mechanism.

Type	Avg. Token # Input	Avg. Token # Output	Avg. $ Cost	Avg. Time (s)
Type-I	20,152	1,057	0.06	53.22
Type-II	31,831	2,180	0.10	100.25
Type-III	57,146	4,169	0.19	193.66
Type-IV	89,804	4,776	0.27	222.52
Total	60,649	3,589	0.19	166.58

TABLE 4: **Average Time & Money Cost of PORTGPT.**

tool invocations. Moreover, while GPT-5 consistently executes tool calls accurately, Llama 3.3 frequently generates function-calling messages with incorrect formats or parameters, severely compromising its effectiveness. The Berkeley Function Calling Leaderboard [67] further validates these other models' superior performance compared to Llama 3.3 in function-calling tasks. While DeepSeek-v3 demonstrates better performance in tool invocation (compared to Llama 3.3), its accuracy still falls short of the other models due to its limitations in code comprehension and contextual understanding.

6.3. Ablation Study (RQ2)

We conducted an ablation study, selectively disabling each of PORTGPT's well-designed tools to measure their impact on patch backporting performance. With each case requiring four evaluations under this protocol, testing a dataset of approximately 2000 cases leads to prohibitive monetary costs, estimated in the thousands of dollars. Therefore, to quantify each tool's contribution to the LLM's backporting capabilities under these constraints, our analysis only utilized our dataset.

The results, shown in Table 3, highlight the contributions of each tool. Firstly, we assess *GitUtils* and *LocateSymbol* tools that contribute to patch localization. Though these two tools cooperate to achieve better performance, they could also work independently. Disabling *GitUtils* caused a sharp drop in overall performance, particularly in Type-IV cases, where the accuracy plummeted to 32.05%. This outcome underscores the importance of *GitUtils* for handling complex relationships between commits. Similarly, removing the *LocateSymbol* tool led to reduced performance, especially in Type-III cases, where accuracy fell to 69.39%, demonstrating its critical role in identifying symbol locations. Removing both tools simultaneously results in the most severe degradation, with overall accuracy dropping to 44.52%, confirming their synergistic effect in patch localization.

Secondly, we assess AutoFix mechanism in *ApplyHunk* and *CompileTest* tools that contribute to patch transformation. Disabling AutoFix caused performance to degrade across all types, with Type-III and Type-IV cases being particularly affected, showing accuracies of 65.31% and 30.77%, respectively. This result underscores AutoFix's critical role in refining intermediate patch quality. Removing *CompileTest* also led to performance drops across all case types, reflecting its importance in final patch combination and integration. Overall, the complete system, PORTGPT, achieves the best performance, excelling in challenging scenarios such as Type-IV cases with 43.59% accuracy and achieving a total accuracy of 62.33%. These results confirm that the integration of all components is essential for optimal system performance.

6.4. Efficiency Evaluation (RQ3)

The efficiency of PORTGPT is evaluated on the whole dataset in terms of average token usage, monetary cost, and processing time, as shown in Table 4. Across different input-output scenarios (Type-I to Type-IV), the average time and cost scale proportionally with the complexity of the task, with larger input and output sizes leading to higher resource consumption. As the complexity of patches increases (from Type-I to Type-IV), the LLM requires more information queries and performs more reasoning, leading to higher costs. The total average input token count is 60,649, producing an average of 3,589 output tokens, with a cost of $0.19 and a processing time of 166.58 seconds. This balance between cost-effectiveness and processing speed demonstrates the practical scalability of PORTGPT for various levels of task complexity, maintaining reasonable efficiency even for the most demanding scenarios.

6.5. Real-World Applicability Study (RQ4)

To evaluate the practical applicability of PORTGPT, we conducted studies across two prevalent and challenging real-world backporting scenarios: (1) porting patches from the mainline Linux kernel to a LTS version, and (2) propagating patches from an LTS version to downstream distributions. All patches selected for these scenarios were introduced after the knowledge cutoff of the underlying LLM (GPT-4o), allowing for a robust assessment of PORTGPT's generalization to unseen tasks.

Mainline To LTS. For this scenario, our selection process focused on challenging CVE patches (2024-2025, sourced from Linux vulns [58])—specifically those that had previously failed backporting from mainline to Linux 6.1-stable and lacked manual conflict resolution. This rigorous filtering yielded a focused set of just 18 such cases. PORTGPT successfully handled 9 of these (50%). Critically, all 9 generated

patches were subsequently reviewed and merged into the 6.1-stable branch by kernel maintainers. For the remaining 9 cases, 7 failed due to significant structural code changes, and 2, though directly cherry-pickable, exhibited regressions post-application [22]. In contrast, TSBPORT handled only 2/18 cases (11.1%), underscoring PORTGPT's improved applicability for complex mainline-to-LTS backports.

LTS To Downstream. We evaluated PORTGPT on 16 patch pairs (10 CVEs) for Ubuntu (targeting Jammy, Focal, Bionic versions) [56]. All patches were introduced after October 2023 to test generalization when porting from LTS sources to downstream targets. PORTGPT successfully backported 10 out of 16 tasks (62.50%)(details in Table 6 in Appendix). This demonstrates PORTGPT's robust generalization for LTS-to-downstream backporting of unseen patches.

6.6. Case Studies

In this part, we present two cases to illustrate how the design of PORTGPT effectively supports LLM reasoning.

CVE-2020-22030. In both versions, the patch introduces a length check before usage as the core fix. However, as code evolves across versions, the function and struct names, as well as their usages in the patch, have changed. As a result, beyond resolving contextual conflicts, the added logic in the patch also requires careful adaptation to align with the target version's codebase, which poses a significant challenge for automated backporting tools. Specifically, the function `ff_inlink_queued_samples` was replaced by `ff_framequeue_queued_samples`, and the field `inputs` was used differently. In this case, the *History* component in PORTGPT's *GitUtils* module effectively provided relevant historical commit information to the LLM, enabling it to reason about and adapt the necessary changes. In contrast, due to the lack of historical context, TSBPORT reused the mainline implementation without adapting to version-specific changes, leading to an incorrect backport.

CVE-2023-41175. The added lines in the patch also require adaptation to fit the target version. Specifically, the macro `UINT_MAX` does not exist in the target version. However, this issue cannot be inferred from the patch's code context or its historical modifications, and can only be revealed through compilation and testing. TSBPORT, which performs no post-generation verification (including compilation), produces a "seemingly correct" patch that ultimately fails. In contrast, PORTGPT adopts a two-stage design that effectively eliminates hard-to-infer transformations during backporting. Its iterative process, compilation testing and patch correction based on observed errors, also closely aligns with real-world backporting practices.

7. Discussion and Limitations

In this section, we discuss the limitations of PORTGPT.

Results Interpretation. While PORTGPT demonstrates promising results, it currently cannot consistently provide reliable interpretations of backported patches to explain their correctness. This limitation arises because LLMs are statistical models of syntactic representation. Although they can generate syntactically correct code, they lack the ability to provide a robust semantic understanding. Therefore, we recommend that PORTGPT users treat backported patches and outputs as suggestions, carefully reviewing the patches manually before applying them to the target codebase.

Context Length. One of the key limitations of LLMs is their restricted context length, which can hinder performance when the input exceeds a certain size. Research [40] has shown that the effectiveness of LLMs decreases as the length of the input increases, due to their inability to process long-range dependencies effectively. In PORTGPT, this issue is mitigated by processing the hunks of the original patch separately during the first stage, allowing each part to be handled within the context length constraints. However, this strategy comes with its own challenges, as it can result in a loss of crucial context from the original patch, potentially affecting the accuracy of the backporting process.

8. Related Work

Program Repair. Automated program repair (APR) aims to automatically fix bugs in software systems, sharing a goal similar to backporting but operating without access to the original patch. Among the various tasks in APR, patch generation has received the most attention. This task involves taking a buggy code snippet as input and producing a patch to fix the bug. Many approaches leveraging deep learning [8], [31], [34] and large language models (LLMs) [60], [64], [65], [73] have shown promising results in this area. A prerequisite task for APR is fault localization (FL), which identifies the buggy code snippet. This process resembles PORTGPT, where LLMs are used to determine the corresponding patch location of a hunk. However, FL in APR is often more challenging due to the absence of the original patch. Current FL methods are predominantly statistical [4], [46], [50], [66], scoring elements in the program to analyze root causes. Another critical step in APR is patch validation, which is typically conducted through either manual review [62] or automated testing [47]. In this work, we primarily rely on manual efforts to validate the backported patches.

LLM for SE. LLM have become powerful tools for various software engineering (SE) tasks, such as vulnerability detection [29], [53], code translation [38], [44], program repair [1], [30], [47], [64], and unit test generation [2], [3]. To assess their effectiveness in real-world applications, benchmarks like SWE-bench [37] have been developed. SWE-bench consists of many GitHub issues and their corresponding pull requests from widely-used Python repositories, providing a challenging dataset for evaluating how well models can generate patches to resolve specific codebase issues. Several notable tools have been evaluated using SWE-bench. SWE-Agent [69] leverages LLMs to autonomously tackle

GitHub issues, utilizing a custom Agent-Computer Interface (ACI) to improve the model's ability to navigate, edit, and test code within repositories. Similarly, AutoCodeRover [74] integrates LLMs with advanced code search mechanisms, leveraging program structures such as abstract syntax trees to enhance bug fixing and feature implementation. Building on strong capabilities of LLMs, our work uses these models to facilitate patch backporting between different versions.

9. Conclusion

Patch backporting is a complex and labor-intensive task in the real world, which increases the burden of project maintainers. In this work, we introduced PORTGPT, an LLM-based tool designed for end-to-end automated patch backporting. Using the modification history effectively in Git, PORTGPT employs a combined approach to achieve precise localization, guiding the LLM to make inferences based on the provided information, thus simulating the expert backporting process. Our extensive evaluation on diverse datasets demonstrate that PORTGPT significantly outperforms existing automated patch backporting tools. Evaluations on multi-language, multi-project, and more complex patch types also demonstrate PORTGPT's ability to handle diverse scenarios and its versatility.

References

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215*, 2023.

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. *arXiv preprint arXiv:2402.09171*, 2024.

[3] Nadia Alshahwan, Mark Harman, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Assured llm-based software engineering (keynote paper). In *2nd. ICSE workshop on Interoperability and Robustness of Neural Software Engineering (InteNSE)(Lisbon, Portugal). To appear*, 2024.

[4] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. {AURORA}: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Debasish Chakroborti, Kevin A. Schneider, and Chanchal K. Roy. Backports: change types, challenges and strategies. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2022.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2022.

[9] CVE-2019-16232 Mainline Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7da413a18583baaf35dd4a8eb414fa410367d7f2.

[10] CVE-2021-4197 Mainline Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1756d7994ad85c2479af6ae5a9750b92324685af.

[11] CVE-2021-4197 Stable Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a70bcf9ed08f3628a9324f054b0e041697b26853.

[12] CVE-2022-32250 Mainline Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=520778042ccca019f3ffa136dd0ca565c486cedd.

[13] CVE-2022-32250 patch for mainline failed to apply in 5.4-stable. https://lore.kernel.org/stable/1654262711245226@kroah.com/.

[14] CVE-2022-41720 backported patch in golang v1.19 branch. https://github.com/golang/go/commit/d80340177116c079fb2ad681dd4aaa4bdc27b770.

[15] CVE-2022-41720 original patch in golang master branch. https://github.com/golang/go/commit/7dc9fcb13de7bb20b11f6a526865545cc9142c2c.

[16] CVE-2023-24023 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=04a342cc49a8522e99c9b3346371c329d841dcd2.

[17] CVE-2023-51779 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2e07e8348ea454615e268222ae3fc240421be768.

[18] CVE-2023-52752 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d328c09ee9f15ee5a26431f5aad7c9239fa85e62.

[19] CVE-2024-0565 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=eec04ea119691e65227a97ce53c0da6b9b74b0b7.

[20] CVE-2024-24860 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da9065caa594d19b26e1a030fd0cc27bd365d685.

[21] CVE-2024-25742 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e70316d17f6ab49a6038ffd115397fd68f8c7be8.

[22] Cve-2024-26889 patch revert. https://lore.kernel.org/stable/20241115063722.766718123@linuxfoundation.org/.

[23] CVE-2024-26922 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6fef2d4c00b5b8561ad68dd2b68173f5c6af1e75.

[24] CVE-2024-41066 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0983d288caf984de0202c66641577b739caad561.

[25] CVE-2024-43863 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e58337100721f3cc0c7424a18730e4f39844934f.

[26] CVE-2024-46743 Patch in Linux. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b739dffa5d570b411d4bdf4bb9b8dfd6b7d72305.

[27] DeepSeek-AI. Deepseek-v3 technical report, 2024.

[28] Detailed Description of Unified Format Patch. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html.

[29] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

[30] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[31] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 935–947, 2022.

[32] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

[33] GPT-4o. https://platform.openai.com/docs/models#gpt-4o.

[34] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[35] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. Saver: Scalable, precise, and safe memory-error repair. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

[36] Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/.

[37] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[38] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[39] LangChain. https://www.langchain.com/.

[40] Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848*, 2024.

[41] Linux kernel release lifecycle. https://www.kernel.org/category/releases.html.

[42] Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37:81857–81887, 2024.

[43] Node.js release lifecycle. https://nodejs.org/en/about/previous-releases.

[44] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. *arXiv preprint arXiv:2308.03109*, 2023.

[45] Shengyi Pan, You Wang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. Automating zero-shot patch porting for hard forks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 363–375, 2024.

[46] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. Benzene: A practical root cause analysis system with an under-constrained state mutation. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2023.

[47] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[48] Ridwan Shariffdeen, Xiang Gao, Gregory J Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. Automated patch backporting in linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 633–645, 2021.

[49] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. Automated patch transplantation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(1):1–36, 2020.

[50] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[51] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1993–2010, 2022.

[52] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 1981.

[53] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

[54] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. Understanding the practice of security patch management across multiple branches in oss projects. In *Proceedings of the ACM Web Conference 2022 (WWW)*, 2022.

[55] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, page 1630–1644, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Ubuntu. https://ubuntu.com/.

[57] universal-ctags. https://github.com/universal-ctags/ctags.

[58] Linux vuls.git. https://git.kernel.org/pub/scm/linux/security/vulns.git/.

[59] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. Core: Resolving code quality issues using llms. *Proceedings of the ACM on Software Engineering*, 1(FSE):789–811, 2024.

[60] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[61] Susheng Wu, Ruisi Wang, Yiheng Cao, Bihuan Chen, Zhuotong Zhou, Yiheng Huang, JunPeng Zhao, and Xin Peng. Mystique: Automated vulnerability patch porting with semantic and syntactic-enhanced llm. *Proceedings of the ACM on Software Engineering*, 2(FSE):130–152, 2025.

[62] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. Mitigating security risks in linux with {KLAUS}: A method for evaluating patch correctness. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[63] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[64] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[65] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[66] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the negative force: Efficient vulnerability {Root-Cause} analysis through reinforcement learning on counterexamples. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[67] Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Berkeley function calling leaderboard. 2024.

[68] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 560–573, 2025.

[69] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

[70] Su Yang, Yang Xiao, Zhengzi Xu, Chengyi Sun, Chen Ji, and Yuqing Zhang. Enhancing oss patch backporting with semantics. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2366–2380, 2023.

[71] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 2024.

[72] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[73] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. Patchagent: A practical program repair agent mimicking human expertise. In *34rd USENIX Security Symposium (USENIX Security 25)*, 2025.

[74] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024.

[75] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Appendix

1. Illustration of AST-based backporting

Figure 4 illustrates FIXMORPH's context matching process using AST. Even though this example involves the renaming of functions and variables, their underlying syntactic structure remains identical at the AST level. This allows FIXMORPH to effectively handle such superficial textual variations within the code context.

Figure 5, on the other hand, depicts FIXMORPH's transformation stage. In this visualization, red highlights AST structures identified as identical and thus marked for deletion, while green denotes newly added AST structures. Finally, nodes corresponding to variables that have undergone changes are updated with their new names (e.g., `buf`).

2. User Prompt of PORTGPT

Figure 6 details the prompts designed for the LLM specifically for patch backporting tasks. For the two identified stages of this process, we have established distinct workflows for the LLM. These initial prompts direct the LLM to invoke appropriate tools according to the specified workflow and subsequently analyze the resulting outputs.

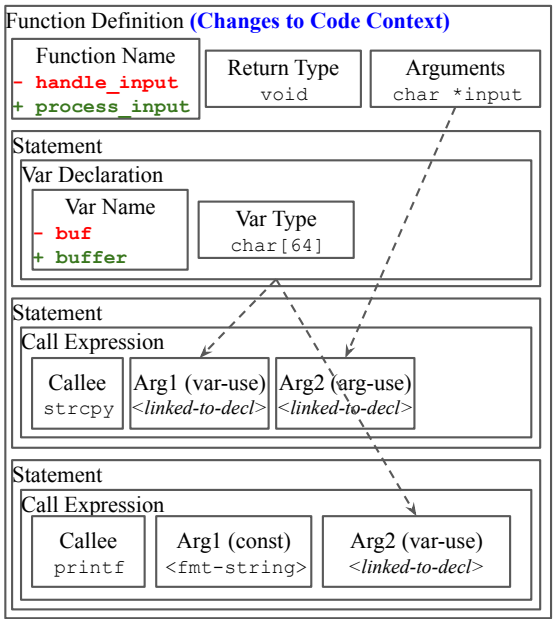

Figure 4: The AST representation of how L_o changes to L_n around ΔP in Listing 2: two symbol renamings.

3. Dataset

Table 5 presents an overview of the dataset collected by us. It includes the number of patches for each type and the projects from which they were collected. The dataset includes a total of 34 projects. During the patch collection process, we also gathered the compilation scripts required for each patch, to be used in subsequent testing.

4. Ubuntu Backporting Task Results

Table 6 details 16 cases collected from Ubuntu, all originating after GPT-4o's training data cutoff date. We validated the correctness of all these cases against publicly available data from online. Notably, for one patch targeting Bionic (CVE-2024-43863) where online ground truth was unavailable, its correctness was confirmed by Ubuntu maintainers.

5. Failure Case Studies

In this section, We describe the main reasons behind the failures of PORTGPT and present several case studies to illustrate the underlying causes of these failures.

From a high-level perspective, the reasons for failure can be summarized into three points: 1) The target version contains additional vulnerable code compared to the original version, which requires fixing; 2) The hunk-by-hunk backport approach overlooks potential dependencies between hunks. 3) PORTGPT only backports the current patch and does not incorporate its prerequisite commits. The following cases describe these three points in detail.

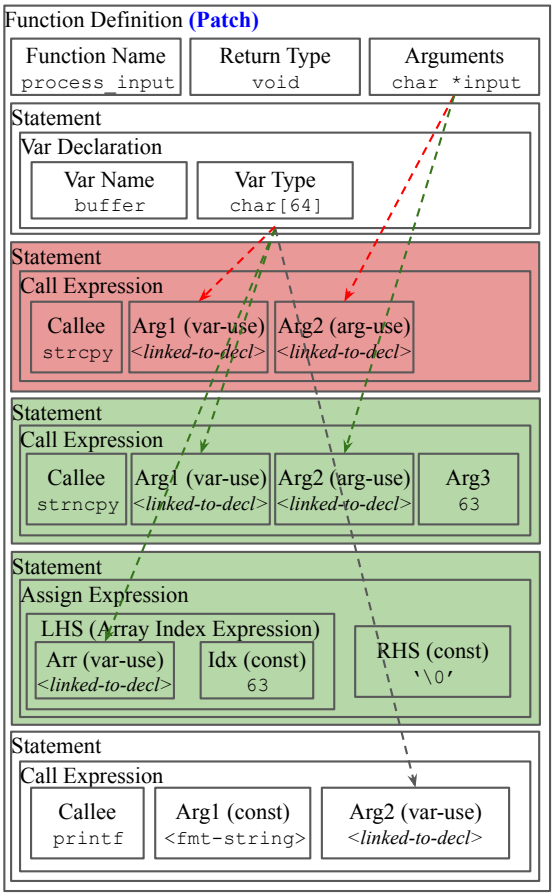

Figure 5: The AST representation of ΔP shown in Listing 2. Red marks deletion while green marks addition.

Stage-1 Prompt

Your **TASK** is to backport a patch fixing a vuln from a original version of the software to an target version step by step. The project is {project_url}. For the ref {original_patch_parent}, the patch below is merged to fix a security issue:
{original_patch}
I want to backport it to ref {target_patch_parent}. To assist you in reviewing the relevant code, I have provided the following code blocks from the target version that closely match the current patch location:
{similar_block}
Your **WORKFLOW** should be: .....

Stage-2 Prompt

Your **TASK** is to validate and revise the patch until it is successfully backported to the target version and really fixes the vulnerability.
Below is the patch you need to backport:
{new_patch}
According to the patch above, I have formed a patch that can be applied to the target version:
{complete_patch}
Now, I have tried to compiled the patched code, the result is: {compile_ret}
You can validate the patch with provided tool *validate*. There are some processes to validate if the patch can fix the vulnerability: .....
If the patch can not pass above validation, you need to revise the patch with the help of provided tools. The patch revision **WORKFLOW** should be: .....

Figure 6: **User Prompt of PortGPT**

CVE-2021-4197 [10], [11]. CVE-2021-4197 highlights a common failure reason in PortGPT's backporting process, as demonstrated in Listing 9. The patch in the original version [10] includes the first hunk, which introduces proper credential handling through `override_creds` and `revert_creds` for `cgroup_attach_permissions`. While PortGPT successfully applies this modification to the corresponding code in the target version, it fails to address a similar vulnerability in the second hunk involving `cgroup_procs_write_permission`. Both code segments require identical security enhancements to handle credentials correctly [11]. The oversight leaves the second vulnerability unpatched and cause PortGPT generates an incomplete patch. In the mainline version, the second segment does not exist since `cgroup_threads_write` is implemented with `__cgroup_procs_write`.

CVE-2019-16232 [9]. CVE-2019-16232 illustrates a failure in PortGPT's backporting process. In the original version, the patch consists of two related hunks: the first introduces the error handling statement `goto err_queue;`, and the second defines the behavior of the `err_queue` label. During backporting, PortGPT replaces `err_queue` with `out` in the first hunk because it cannot find the `err_queue` label in the target version's context. When processing the second hunk, PortGPT directly adds the `err_queue` label as in the original patch, but the label does not integrate correctly because it was not previously established. We believe this failure is due to PortGPT's hunk-by-hunk approach, which fails to account for dependencies between hunks.

CVE-2022-41720 [14], [15]. CVE-2022-41720 is a vulnerability in the Golang project that highlights a limitation in PortGPT 's backporting process. The patch in the master branch [15] revises the `join` function to address the issue; however, this function does not exist in the target version. To address this, the patch for the target version [14] first introduces the `join` function before modifying it. PortGPT fails to backport the patch because it cannot locate the `join` function in the target version and incorrectly assumes that no backporting is needed. This failure stems from PortGPT's inability to follow a common backporting practice where developers include prerequisite commits to adjust the context before applying the actual patch. Addressing this limitation in future work will enable PortGPT to better handle complex backporting scenarios, aligning more closely with experienced developer practices.

Language	Project	Type-II	Type-III	Type-IV	Total
C	Linux	6	4	11	21
	FFmpeg	0	5	7	12
	redis	0	6	4	10
	libtiff	0	3	3	6
	cpython	0	2	2	4
	glibc	0	3	1	4
	gnupg	0	1	2	3
	krb5	0	1	1	2
	libwebp	0	1	1	2
	FreeRDP	0	1	0	1
	OpenSSL	0	0	1	1
	libpng	0	1	0	1
	php	0	1	0	1
	sqlite	0	0	1	1
C++	MongoDB	0	3	7	10
	Chromium	0	2	5	7
	envoy	0	4	1	5
	electron	0	1	1	2
	qt	0	1	1	2
	grpc	0	0	1	1
	v8	0	0	1	1
Go	golang	6	2	7	15
	Argo CD	3	1	7	11
	consul	3	1	2	6
	containerd	0	0	4	4
	nomad	1	1	1	3
	mattermost	0	0	2	2
	moby	0	1	1	2
	Valut	0	1	0	1
	cilium	0	0	1	1
	darp	0	0	1	1
	etcd	0	1	0	1
	go-jose	0	1	0	1
	Grafana	0	0	1	1
Total	/	19	49	78	146

TABLE 5: **Overview of Our Dataset.**

6. Future Directions

To address the limitations outlined above, we propose several promising avenues for future research aimed at enhancing PORTGPT's performance. First, we recommend exploring advanced reasoning techniques [71], [72], as well as leveraging fine-tuning methods [7], to improve the ability of LLMs to handle the intricate dependencies in code. These techniques could help the models better understand the semantic relationships between different parts of the code, leading to more accurate and reliable backported patches. Second, we suggest investigating improved methods for validating the correctness of generated patches. In our current work, we rely on ground truth data to assess the accuracy of the backported patches. However, in real-world scenarios, ground truth data is often unavailable, making it difficult to automate the validation process. Although directed fuzzing is also an effective approach for validating patches, current directed fuzzing techniques are limited in generalizability and cannot be widely applied across diverse programs. We believe that the integration of an automatic validation system could not only streamline this process but also provide valuable feedback that could be used to further refine the LLM's ability to generate high-quality backporting patches.

CVE ID	Version	Fix Date	Type	P.G.
CVE-2024-46743 [26]	Bionic	2024/10/15	II	✓
CVE-2023-51779 [17]	Focal	2024/01/05	III	✓
CVE-2024-0565 [19]	Focal	2024/01/29	III	✓
CVE-2024-26922 [23]	Focal	2024/05/21	III	✓
CVE-2024-43863 [25]	Focal	2024/12/03	III	✓
CVE-2024-43863 [25]	Bionic	N/A	III	✓
CVE-2023-24023 [16]	Focal	2024/03/14	IV	✗
CVE-2023-52752 [18]	Bionic	2024/06/26	IV	✗
CVE-2023-52752 [18]	Focal	2024/06/26	IV	✓
CVE-2023-52752 [18]	Jammy	2024/06/26	IV	✗
CVE-2024-24860 [20]	Bionic	2024/07/09	IV	✗
CVE-2024-24860 [20]	Focal	2024/07/09	IV	✗
CVE-2024-25742 [21]	Jammy	2024/07/02	IV	✗
CVE-2024-26922 [23]	Bionic	2024/05/21	IV	✓
CVE-2024-41066 [24]	Focal	2024/11/25	IV	✓
CVE-2024-41066 [24]	Jammy	2024/11/25	IV	✓

TABLE 6: **Ubuntu Backporting Tasks Results. Version** refers to the target version for Ubuntu backporting tasks. **Fix Date** represents the actual date of the backport in the real world.

```
@@ -4397,8 +4398,15 @@ __cgroup_procs_write(
  spin_unlock_irq(&css_set_lock);
+ saved_cred = override_creds(of->file->f_cred);
  ret = cgroup_attach_permissions(src_cgrp,
  dst_cgrp, of->file->f_path.dentry->d_sb, ...);
+ revert_creds(saved_cred);
  if (ret)
      goto out_finish;

@@ -4440,9 +4449,15 @@ cgroup_threads_write(
  spin_unlock_irq(&css_set_lock);
+ saved_cred = override_creds(of->file->f_cred);
  ret = cgroup_procs_write_permission(src_cgrp,
      dst_cgrp, of->file->f_path.dentry->d_sb);
+ revert_creds(saved_cred);
  if (ret)
      goto out_finish;
```

Listing 9: **CVE-2021-4197.** Both the original [10] and target [11] version patches include the first hunk, but only the target version requires modification of the second hunk.