2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

LLMPORT: Cross-file Patch Porting via Task Decomposition and Self-correction

Bofei Chen[1], Lei Zhang[2], Peng Deng[1], Nan Wang[1],
Haoyu Xu[1], Mingda Guo[1], Yuan Zhang[2], and Min Yang[2]
1: Fudan University, {bfchen22, pdeng21, wangnan24, haoyuxu21, mdguo22}@m.fudan.edu.cn,
2: Fudan University, {zxl, yuanxzhang, m_yang}@fudan.edu.cn

***Abstract*—Security patch porting aims to adapt patches developed for one software version so they can be used in another version. This approach is crucial for maintaining the security of software systems over time. However, existing works often rely on predefined rules to understand patches, limiting their generalizability and portability. Additionally, they are ineffective when porting complex patches that involve numerous modified code lines across multiple files, which is common in real-world software, especially Java applications.**

To overcome these obstacles, we propose a novel patch porting framework, called LLMPORT. First, LLMPORT breaks down the complex patch porting task into distinct subtasks, each containing an atomic code unit from the original patch. This enhances the LLMs' focus. Second, for each subtask, LLMPORT extracts the minimal patch-related code context and constructs a prompt with task-specific domain knowledge to guide the LLM in porting the patch code to the target version. Third, LLMPORT implements a progressive self-correction system to automatically assess the correctness of the generated patch, and identify and correct error subtasks based on LLMs' self-correction capabilities.

We evaluate LLMPORT for porting Java language patches on a large-scale dataset, including 1,992 unique patch file pairs, and it successfully ports 91.92% of them. To assess the portability of LLMPORT, we also evaluate its capability to port C language patches. The results show that it outperforms state-of-the-art approaches, including TSBPORT and FixMorph. LLMPORT also discovers five 0-day vulnerabilities due to incomplete patches and the developers received and merged the new patches generated by LLMPORT into the official code branches.

I. INTRODUCTION

Timely deploying security patches to vulnerable open-source software (OSS) is a primary defense against vulnerability exploitation. However, the code associated with a vulnerability can vary across versions, so developers cannot directly port patches from one version to another, and manual adjustments are very time-consuming and may even introduce errors, especially for large OSS with discrete versions. As a result, developers usually only release security patches on the latest versions for a few branches [29], which forces downstream users to upgrade promptly. However, most downstream users continue to use the vulnerable versions, fearing that the upgrade will break their projects [14], leading to a long-term threat of N-day vulnerabilities. For example, 18 months after the patch release of CVE-2021-44228, it is still affecting 45% of organizations worldwide [8].

Thus, to help those users who do not use the latest OSS, several works have been proposed to port security patches [25], [26], [28], [35] – they aim to adjust the patches developed for the latest OSS and make them usable for other vulnerable releases. For example, FixMorph [25] utilizes local structures and properties to identify patch-related code changes between versions and then uses predefined transformation rules to extract the original code in the original or target version to adjust the patch. TSBPORT [35] proposes four types of predefined rules related to patch patterns for matching and extracting nearby code to adjust patches. However, existing work still fell short in two major limitations, impacting their practicality and effectiveness.

First, existing pattern-based porting approaches [35] [25] rely heavily on reusing original code based on manually defined rules, which limits their generalizability. These rules often fail when there are significant structural or logical changes between software versions, or when relevant code is missing. Second, recent LLM-based approaches typically operate only at the function level, while real-world patches often span multiple code files and involve numerous methods. For example, PPatHF [21] employs a fine-tuned LLM for function-level patch porting across forked projects. Additionally, while LLMs are prone to errors during patch generation, PPatHF does not adequately address the varied causes of these failures, limiting its ability to support further patch adaptation. Third, existing approaches lack mechanisms to handle dependencies between code segments within one patch, a common scenario in patches involving multiple files. This limits their practicality for real-world multi-file patch migration.

To address these gaps, we propose a collaborative approach for porting patches that combines program analysis and large language models (LLMs). Our primary insight is twofold. Firstly, we could leverage the LLM's strong generalization in analyzing complex code and generating comprehension-based code to address the limitations of traditional porting tools that rely on predefined rules. Second, since LLMs are prone to lose focus or hallucinate [17], [22], [27] when porting complex patches that involve extensive modifications and diverse repair logic, we incorporate program analysis techniques to decompose these tasks into smaller sub-tasks with single repair behavior. This enhances the LLM's focus on specific tasks, improving patch migration success.

However, it is not an easy task. First, code refactoring and the intricate dependencies between versions can cause significant differences between the modified statements in a patch and their context in the affected version. This makes it difficult to break down complex patch migrations and extract the essential, minimal context. Second, generic LLMs commonly are short of domain knowledge in security-related tasks. Our study (in § II-A) demonstrates that, when dealing with patches that require structural or logical adjustments, the migration success rate of GPT-4 [9] is only 8.26%. Thus, we should help them understand what patch porting is and how to port patches. To ensure generalizability, we should use abstract domain knowledge to guide LLM in patch porting. However, LLMs are prone to errors when applying this knowledge to concrete complex patches. Additionally, due to the complex dependencies of code modifications in a patch and the inherent stochastic nature of LLMs, their adjustments may only reach a local optimum. Thus, we need to propose a method that allows LLM to globally evaluate the effectiveness of patch adjustment and continuously correct errors.

In this paper, we design a novel security patch porting framework, called LLMPORT (LLM-driven Security Patch Porting), to automatically port a patch from the original version (V_{org})[1] to a target version (V_{tar}). Specifically, LLMPORT contains two main stages. In stage I, LLMPORT introduces a subtask-based approach to decompose complex patch porting tasks into manageable subtasks. This approach focuses on ordered units of the patch with simpler repair logic, which could greatly enhance the LLM's task-specific attention, thereby increasing the success rate of patch porting. For each sub-task, LLMPORT extracts the minimal patch-related code context and constructs a prompt with a specific task description to guide the LLM in generating the corresponding patch code. Furthermore, LLMPORT leverages a *dependency patch graph* to model inter-unit dependencies and guide *N-to-N patch location alignment* under cross-version dependency shifts and evolving vulnerability scopes. Based on this graph, LLMPORT introduces a novel *dependency-aware* mechanism that propagates cross-version patch changes across interdependent units (subtasks) via control-flow, data-flow, and structural links. This mechanism enables coordinated multi-location refinement, extending beyond previous locally scoped adjustment strategies (e.g., TSBPORT). In stage II, LLMPORT proposes an LLM-driven patch correction technique to check and refine the generated patch. It utilizes external tools to identify the error subtask, synthesize fine-grained error feedback, and proposes an on-demand context supplement technique to help LLMs adjust the patch. LLMPORT can ultimately achieve a complete and reliable patch by iteratively adjusting related error subtasks.

We evaluate LLMPORT with four well-known LLMs (e.g., GPT-4 [9], qween [10]) upon a large-scale dataset, which includes 232 CVEs, 1,992 Java patch file pairs[2]. The results show that, LLMPORT could obtain a success rate of 91.92% on our dataset and significantly enhances the performance of LLM-driven security patch porting. For example, compared with out-of-the-box four baseline LLMs, LLMPORT improves 43.18%, 49.95%, 50.15% and 37.40% success rates of patch porting. It also sheds light on porting complex patches. As an example, for patches that have code logic and structural modifications, GPT-4 only successfully port 8.26% of them while LLMPORT could increase the success rate to 73.75%.

We further validate LLMPORT's ability to port patches for recently disclosed vulnerabilities. Out of 69 patches we collected, LLMPORT successfully ported 56 of them, demonstrating the effectiveness of LLMPORT when porting patches not pre-learned by LLMs. Additionally, LLMPORT also identifies five 0-day vulnerabilities (with four CVE IDs assigned) due to incomplete patches, as we find that it tries to fix some other vulnerable code snippets ignored by original patches. We submit these patches to developers and they are all merged into the official code branches, demonstrating the quality of patches generated by LLMPORT.

We also evaluate LLMPORT's capabilities of patch porting tasks on multiple languages (e.g. C/C++). For example, the results show that LLMPORT successfully ported 335 out of 350 patch pairs, significantly outperforming TSBPORT (168 pairs) and FixMorph (246 pairs).

We summarize the contributions of this paper as below:

- We propose a LLM-enhanced patch porting framework LLMPORT. It can effectively port complex patches, validate and correct generated error patches, thereby greatly reducing manual efforts.
- LLMPORT decomposes complex patch porting tasks into smaller and manageable subtasks, enhancing the focus of LLMs, and uses a progressive self-correction system to ensure the validity of the generated patches.
- Our experiment on a large-scale dataset reveals that LLMPORT greatly enhances the ability of LLMs in patch porting. In addition, LLMPORT identifies five 0-day vulnerabilities due to incomplete patches and the patches generated by it were merged by developers.

II. MOTIVATION

LLM-driven patch porting involves employing LLMs to migrate an official security patch (i.e. P_{org}) [3] from the original version (i.e. V_{org}) to another target version (i.e. V_{tar}) which is affected by the same vulnerability. And it ensures that the generated patch addresses the vulnerability and maintains backward compatibility.

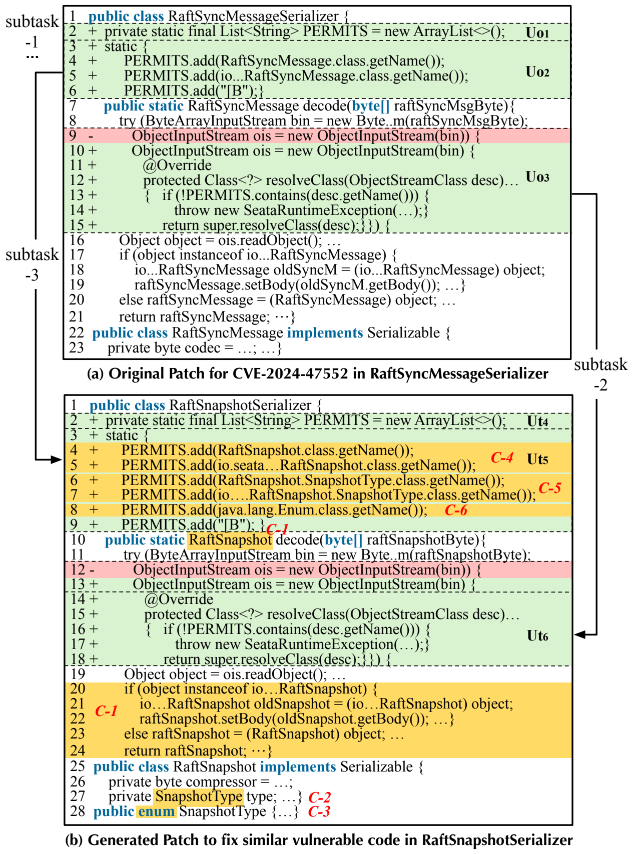
Patch Porting Types. To assess the difficulty of LLM-driven patch porting, we categorize patch porting tasks into four types by considering the definitions of previous researches [25], [35] and the unique difficulties faced by LLMs:

- Type-I (*no changes or only patch location changes*): LLMs only need to find the right patch locations for porting the code in P_{org} to the correct places in V_{tar}.
- Type-II (*deleted code changes*): The deleted code lines in P_{org} need to be adjusted to adapt V_{tar}.

[1] V_a represents the version a of one software.

[2] For Java programs, patches may contain multiple code files. One patch file pair represents the corresponding files in the original OSS version and the target OSS version which should contain the same piece of patch code.

[3] P_a represents the patch on version a of one software.



```
1  public class RaftSyncMessageSerializer {
2  + private static final List<String> PERMITS = new ArrayList<>();
3  + static {
4  +     PERMITS.add(RaftSyncMessage.class.getName());
5  +     PERMITS.add(io...RaftSyncMessage.class.getName());
6  +     PERMITS.add("[B");}
7      public static RaftSyncMessage decode(byte[] raftSyncMsgByte){
8        try (ByteArrayInputStream bin = new Byte..m(raftSyncMsgByte);
9  -         ObjectInputStream ois = new ObjectInputStream(bin)) {
10 +         ObjectInputStream ois = new ObjectInputStream(bin) {
11 +            @Override
12 +            protected Class<?> resolveClass(ObjectStreamClass desc)…
13 +            {   if (!PERMITS.contains(desc.getName())) {
14 +                    throw new SeataRuntimeException(…);}
15 +                return super.resolveClass(desc);}}) {
16       Object object = ois.readObject(); …
17       if (object instanceof io...RaftSyncMessage) {
18          io...RaftSyncMessage oldSyncM = (io...RaftSyncMessage) object;
19          raftSyncMessage.setBody(oldSyncM.getBody()); …}
20       else raftSyncMessage = (RaftSyncMessage) object; …
21       return raftSyncMessage; ⋯}
22  public class RaftSyncMessage implements Serializable {
23    private byte codec = …; …}
```

(a) Original Patch for CVE-2024-47552 in RaftSyncMessageSerializer

```
1  public class RaftSnapshotSerializer {
2  + private static final List<String> PERMITS = new ArrayList<>();
3  + static {
4  +    PERMITS.add(RaftSnapshot.class.getName());
5  +    PERMITS.add(io.seata…RaftSnapshot.class.getName());
6  +    PERMITS.add(RaftSnapshot.SnapshotType.class.getName());
7  +    PERMITS.add(io.…RaftSnapshot.SnapshotType.class.getName());
8  +    PERMITS.add(java.lang.Enum.class.getName());
9  +    PERMITS.add("[B"); }
10    public static RaftSnapshot decode(byte[] raftSnapshotByte){
11      try (ByteArrayInputStream bin = new Byte..m(raftSnapshotByte);
12 -       ObjectInputStream ois = new ObjectInputStream(bin)) {
13 +       ObjectInputStream ois = new ObjectInputStream(bin) {
14 +          @Override
15 +          protected Class<?> resolveClass(ObjectStreamClass desc)…
16 +          {   if (!PERMITS.contains(desc.getName())) {
17 +                  throw new SeataRuntimeException(…);}
18 +              return super.resolveClass(desc);}}) {
19      Object object = ois.readObject(); …
20      if (object instanceof io…RaftSnapshot) {
21         io…RaftSnapshot oldSnapshot = (io…RaftSnapshot) object;
22         raftSnapshot.setBody(oldSnapshot.getBody()); …}
23      else raftSnapshot = (RaftSnapshot) object; …
24      return raftSnapshot; ⋯}
25  public class RaftSnapshot implements Serializable {
26    private byte compressor = …;
27    private SnapshotType type; …}
28  public enum SnapshotType {…}
```

(b) Generated Patch to fix similar vulnerable code in RaftSnapshotSerializer

Figure 1: The official patch for CVE-2024-47552 and LLMPORT-generated patch to address this incomplete patch

- Type-III (*namespace changes*): LLMs should change identifiers (e.g., method name) in P_{org} to adapt V_{tar}.
- Type-IV (*logic and structural changes*): Further adjustments are needed to adapt P_{org} for V_{tar}. For example, use different codes to convey the same functionality, remove redundant or add missing code changes.

A. A Motivating Example

Figure 1 (a) shows the simplified security patch for CVE-2024-47552, which addresses a Java deserialization vulnerability in Apache Seata [1] on $V_{2.2.0}$. The patch overrides the `resolveClass` method to add a sanity check at Line 13. This check prevents the deserialization of untrusted classes outside the whitelist (Lines 2–6), thereby safeguarding seata against deserialization attacks. However, CVE-2024-47552 affects multiple versions of Seata, e.g., $V_{2.0.0}$. But the developers only implement a patch for $V_{2.2.0}$. Thus, we should port the P_{org} from $V_{2.2.0}$ to $V_{2.0.0}$ to protect their downstream users.

We begin by identifying vulnerable code in $V_{2.0.0}$ that corresponds to the security issue addressed by P_{org}. For example, for the patched method `decode` shown in Figure 1 (a), we identify two `decode` methods that exhibit similar vulnerability code in $V_{2.0.0}$, i.e., within classes RaftSyncMessageSerializer and RaftSnapshotSerializer. We then fix the vulnerable code in V_{tar} using P_{org} as a reference. We submitted the LLMPORT-generated patch to the developers, and they merged them to fix the vulnerable RaftSnapshotSerializer in $V_{2.2.0}$, demonstrating the effectiveness of LLMPORT.

However, porting P_{org} in Figure 1 (a) to RaftSnapshotSerializer in Figure 1 (b) poses challenges, primarily because it requires meticulous adaption of three code segments—a task currently infeasible with existing tools or out-of-the-box LLMs. For instance, it needs first to analyze the differences in `decode` method related to the newly added sanity check (Line 16 in Figure 1 (b)), specifically focusing on changes in the handling of deserialized objects, i.e., *C-1* (Lines 20–24 in Figure 1(b) and Lines 17–21 in Figure 1 (a)). Then, it should adapt class names in the whitelist (i.e., `PERMITS`) to maintain functionality (i.e., *C-4*), such as replacing class `RaftSyncMessage` with `RaftSnapshot`. Additionally, due to discrepancies between the fields of the `RaftSyncMessage` and `RaftSnapshot` classes, the LLM must implement two successor adjustments to ensure compatibility with Java's deserialization mechanism, which mandates that all fields in `RaftSnapshot` be deserializable. Specifically, it needs to realize the `RaftSnapshot` class introduces a new field of type `SnapshotType` (*C-2*), defined as an `enum` (*C-3*), and adds these class names in `PERMITS` accordingly, i.e., *C-5–6*.

Limitations of existing approaches. We now use this example to illustrate the limitations of existing approaches. Firstly, pattern-based approaches rely heavily on manually defined rules to interpret and adapt patches based on the original code, which limits generalizability. For example, TSBPORT attempts to refine backported patches by extracting surrounding code logic using predefined rules. However, in cases like *C-4* to *C-6*, no relevant logic exists in the original open-source version, rendering this strategy ineffective. Moreover, TSBPORT fails to recognize the need to adapt whitelists for different deserialized object types across modules (e.g., *C-1*), leading to incompatibilities with existing business functionality. Second, recent LLM-based methods target single-function patches, ignoring broader context, thus failing at this complex multi-file migrations.

Secondly, they lack the abilities to handle the dependencies of different code segments in one patch, which is common for complex patches, especially for those containing multiple patch-related files. As a result, these approaches overlook the correlation between cross-file patch-related codes and patch changes, failing to establish links induced by the functional semantics of the Java native deserialization protocol between *C-1* and *C-4*, *C-4* and *C-2* as well as *C-2* and *C-5*, etc. And they even require a known target patch file before porting and only support migration between single files (e.g., TSBPORT).

Limitations of out-of-the-box LLMs. To assess whether out-of-the-box LLMs are sufficient for direct use in security patch porting, we evaluate their effectiveness on large-scale porting tasks. Specifically, we constructed a dataset of 766 Java patches with 1,992 unique patch file pairs. We selected four representative LLMs and asked each LLM to generate a P_{tar} for V_{tar} using P_{org} as a reference. Then, we verified the quality of the generated P_{tar}s manually. Table I shows the overall results. The LLMs performed well on simpler patches, showing their ability to identify the location of modified code from a limited patch-related context. For example, GPT-4 successfully ported 61.61% of the Type-I patches. However, as the porting difficulty increases, the success rate of LLMs significantly decreases. Even the best-performing GPT-4 manages to successfully port only 35.33%, 18.07%, and 8.26% of Type II-IV cases, respectively. Evidently, out-of-the-box LLMs demonstrate notable limitations, as further discussed in § II-B.

B. Understanding Challenges

Challenge #1: LLMs tend to lose focus when porting complex patches. First, patches often contain many code modifications with diverse repair logic, making it difficult for LLMs to capture critical contexts and maintain focus on intricate relationships between patch-related code. For example, LLMs often miss the modifications at Lines 4–5 in Figure 1 (b), as they overlook *C-1* and fail to adapt *C-4* accordingly. Therefore, it is essential to decompose the porting task into manageable subtasks, though this is not easy. Specifically, each subtask consists of a code unit of P_{org} and the location where it should be ported in V_{tar}. However, the implementation of the same code logic could change significantly in different software versions, requiring a complex *N-to-N mapping* for identifying the exact locations for each code modification. This makes simple syntactic similarity ranking-based methods [35] less reliable for identifying the target location precisely.

Second, LLMs can easily lose focus or hallucinate if given too much or too little context. To improve prompting, we should extract the *minimal patch-related code context* across versions for each subtask, capturing all necessary and non-redundant code. For example, we should identify *C-1* in the target vulnerable code and determine that the `RaftSyncMessage` class should be replaced with the `RaftSnapshot` class. Furthermore, we need to extract fields with different types between these two classes to guide LLMs in recognizing differences *C-2* and *C-3*. However, we cannot supply redundant context as LLMs are easily distracted [27].

Challenge #2: LLMs are prone to making mistakes and errors can accumulate in the subtask mechanism. This makes it difficult to guarantee the quality of ported patches.

First, LLMs lack domain knowledge for patch porting, making them prone to mistakes, particularly in complex tasks that involve multiple steps. For instance, even when provided with changes to the return type of the `decode` method and *minimal patch-related code context*, GPT-4 focuses only on the direct code difference (i.e., *C-1*). It fails to infer the dependencies between subsequent changes (*C-2–3*) and cannot reliably adapt *C-5* and *C-6* to ensure successful deserialization of all fields within a RaftSnapshot instance.

Second, to ensure the generalization of LLMs, we should avoid providing them with overly specific domain knowledge about the current patch porting task. Patch-related contexts vary greatly between CVEs or OSS, making it difficult for LLMs to apply abstract knowledge to concrete patch porting tasks. The random nature of LLM may also lead to unpredictable errors. Thus, error-checking and correction mechanisms are needed to help LLMs apply generalized knowledge to specific complex instances. However, due to the complexity and variety of potential errors, designing a universal solution to correct these errors is challenging.

C. System Architecture

To address the above challenges, we design a novel security patch porting approach, called LLMPORT, to automatically port a patch from V_{org} to V_{tar}. The overall architecture of LLMPORT is illustrated in Figure 2, and its workflow is detailed in Algorithm 1. It operates in two principal stages: (i) modeling the step-by-step patch migration and creating prompts for each porting (§ III); (ii) auditing the generated patch and correcting it based on error feedback (§ IV).

In stage I, LLMPORT first decomposes the P_{org} into several patch units based on patch pattern and captures their dependencies for step-by-step porting. Second, LLMPORT employs a dependency-aware matching method to align patch contexts across versions, including each patch unit's location in V_{tar} and the context of modified statements with dependencies (e.g., control/data flow), as well as related code differences across versions. It also supports complex N-to-N patch mappings arising from varying code dependencies between versions. Based on above information, LLMPORT extracts the minimal patch-related code context and constructs a task-specific prompt for each subtask to better guide the LLM in generating the corresponding patch code.

In stage II, LLMPORT first invokes external gadgets such as a compiler to audit the generated patch and locates the erroneous subtasks. It then performs static program analysis to help infer the cause of errors and adjust the prompts. Second, before the LLM corrects the patch, LLMPORT performs on-demand patch-related context supplement to guide the LLM in rethinking and making more accurate corrections. LLMPORT iteratively performs the aforementioned two steps to correct the generated patch to ensure all errors are well handled.

III. SUBTASK-BASED PATCH PORTING

Each patch often contains numerous code modifications and affects multiple files, especially in large, feature-rich Java applications. To focus the LLMs' attention, we first slice the patch into small code units and generate a specific sub-porting task for each unit. Generally, a subtask contains a piece of official patch code, the location where it should be placed in V_{tar}, and a hint of potential code changes across versions.

A. Slicing Patches into Atomic Units

Here contains three substeps. First, LLMPORT decomposes P_{org} into smaller, atomic units, each representing a specific and distinct repair action. This process aims to isolate individual changes to avoid semantic breaks, ensuring that each unit retains its intended repair function without affecting others.

Table I: The success rate of out-of-the-box LLMs in porting 707 Java patch pairs and 1,992 patch file pairs.

			Patch File Pair					
	LLM	Context Window	Type-I	Type-II	Type-III	Type-IV	Total	Patch Pair
# Our Dataset			1,420 (71.29%)	150 (7.53%)	83 (4.17%)	339 (17.02%)	1,992	707
# The Results	GPT-4	128,000 tokens	875 (61.61%)	53 (35.33%)	15 (18.07%)	28 (8.26%)	971 (48.74%)	311 (43.99%)
	GPT-3.5	16,385 tokens	473 (33.31%)	31 (20.67%)	9 (10.84%)	20 (5.90%)	533 (26.76%)	201 (28.42%)
	qwen	6,000 tokens	625 (44.01%)	33 (22.00%)	10 (12.05%)	13 (3.83%)	681 (34.19%)	271 (38.33%)
	gemma	8,192 tokens	289 (20.35%)	11 (7.33%)	0 (0%)	5 (1.47%)	305 (15.31%)	130 (18.39%)

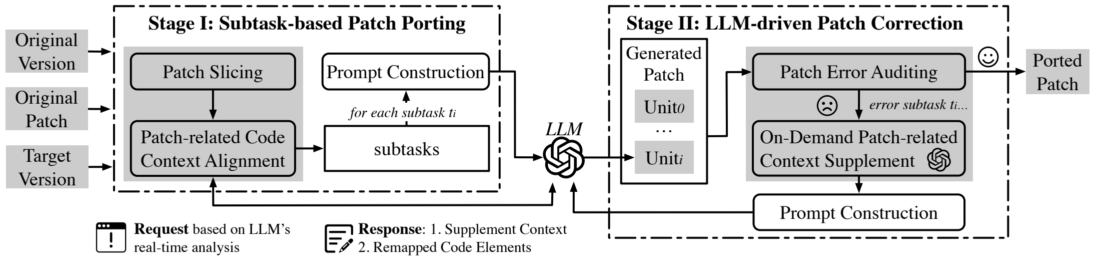

Stage I: Subtask-based Patch Porting
Original Version
Original Patch
Target Version
Patch Slicing
Patch-related Code Context Alignment
Prompt Construction
for each subtask ti
subtasks
LLM
Stage II: LLM-driven Patch Correction
Generated Patch
Unit0
...
Uniti
Patch Error Auditing
error subtask ti...
On-Demand Patch-related Context Supplement
Prompt Construction
Ported Patch
Request based on LLM's real-time analysis
Response: 1. Supplement Context 2. Remapped Code Elements

Figure 2: The Architecture of LLMPort

Algorithm 1 LLMPort(P_{org},V_{org},V_{tar})

```
Input: Official patch $P_{org}$ on a patched version $V_{org}$, target vulnerable version $V_{tar}$
Output: Patched project of the target version $V_{patched}$
1: $U \leftarrow$ decompose($P_{org}, V_{org}$) // Slicing the patch into atomic units
2: $PG \leftarrow$ buildPatchGraph($U$) // nodes are units; edges encode control, data, and structural dependencies
3: $S \leftarrow \varnothing$
4: for all $u \in$ bfsOrder($PG$) do
5:     $C_u \leftarrow$ collectDependencies($u$, $PG$)
6:     $S \leftarrow S \cup$ generateSubtasks($u, C_u, V_{org}, V_{tar}$) // N-to-N patch unit context alignment and generation of patch porting subtasks
7: end for
8: $P_{tar} \leftarrow \varnothing$ // Generated patch for $V_{tar}$
9: for all $s \in$ order($S$) do
10:    $D_s \leftarrow$ getDominators($s, PG$) // Retrieve dominator units that must be applied prior to $s$
11:    $p \leftarrow$ generatePatch($s, D_s, V_{org}, V_{tar}, \varnothing$) // Produce an initial patch unit
12:    while isDuplicate($p, P_{tar}$) $\wedge$ $\neg$exceedRetries() do
13:        $p \leftarrow$ generatePatch($s, D_s, V_{org}, V_{tar}$, errorReport) // Re-generating the patch unit to fix errors from overwriting non-vulnerable code or duplicate modification locations
14:    end while
15:    $(P_{tar}, V_{patched}) \leftarrow (P_{tar}, V_{patched}) \cup$ applyPatch($p, V_{tar}$)
16: end for
17: $(\mathcal{E}, \mathcal{F}) \leftarrow$ auditPatch($V_{patched}$) // Run compiler, static analysis, and test suites to detect errors and failing cases. $\mathcal{E}$: error subtask, $\mathcal{F}$: error feedback.
18: while $\mathcal{F}$ $\wedge$ $\neg$exceedRetries() do
19:    $summary \leftarrow$ summarizeHistory($\mathcal{E}, \mathcal{F}$) // Summarize the adjustment history for the current error task
20:    $D_{\mathcal{E}} \leftarrow$ getDominators($\mathcal{E}, PG$)
21:    for all $d \in D_{\mathcal{E}}$ do
22:        $summary \leftarrow summary \cup$ summarizeDiffs($d$) // Summarize the patch-related context differences associated with the current subtask
23:    end for
24:    $(P_{tar}, V_{patched}) \leftarrow$ correctPatch($\mathcal{E}, \mathcal{F}, summary, V_{org}, V_{patched}$) // Leveraging the LLM's self-correction ability to correct incorrect patch
25:    $(\mathcal{E}, \mathcal{F}) \leftarrow$ auditPatch($V_{patched}$)
26: end while
27: return $V_{patched}$
```

Second, LLMPort parses important code elements, such as newly defined variables, within each patch unit to understand the specific repair behavior. These code elements also help identify the dependencies between units and indicate direct or indirect patch-related code differences across versions. Third, LLMPort identifies the dependencies between individual patch units and groups dependent units into sub-porting tasks.

Patch Unit Decomposition and Parsing. To break down P_{org} into atomic units, LLMPort adopts a generalized patch pattern to slice the patch, ensuring it is not restricted to specific vulnerability types. As demonstrated in Table II (a), we consider nine representative types of patch patterns that encapsulate different high-level semantics of how the modified code addresses vulnerabilities. This categorization aligns with existing research [31], [34], [35], [37] and also takes into account the features of Java language, e.g., `class`. Through our evaluation, the first eight patterns cover 91.07% of modified code blocks in our benchmark.

Concretely, LLMPort first uses the diff [5] tool to extract the modified code blocks in P_{org}. It then utilizes the Tree-sitter [6] query language, along with customized parsing rules to identify the patch patterns of modified statements in each block. Next, it slices the whole patch into different atomic units based on the identified patterns, and merges these units that intersect within one statement or method. For modified statements that do not match any pattern, it treats successive ones as an unit. For example, for code lines 2–6 (Figure 1), LLMPort generates two units, including Line 2 ("Add Variable Declaration"), and Lines 3–6 ("Others").

After analyzing our benchmark, we found that 51.34% of cross-version patch code adjustments are closely related to changes in the invoked/declared methods, defined/used variables, and instantiated/extended classes in the modified code. Thus, LLMPort compares the added and deleted statements with the same pattern in each unit, identifies the specific modified code elements, and extracts their properties. For example, for the patch unit in lines 3–6 of Figure 1 (a), LLMPort extracts three code elements, e.g., class `RaftSyncMessage`, and parses its properties defined in Table II (b), e.g., its fields.

Handling Unit Dependencies. A subtask involves a small step within the overall patch porting task, such as porting one atomic patch unit. However, since dependencies exist between patch units, we should first group them and then port them sequentially. Thus, we link the generated subtasks to group a

Table II: The categorization of patch patterns and code elements.

	Type	Description
	Modify Variable Values	(change/add/delete) the value of the variable $[x]$ to change the program state
	Modify Variable Declaration	(change/add/delete) the declaration/definition statements of (variable/field) $[x]$ (to modify $[x]$'s $[properties]$, e.g., access modifier)
	Modify Class Declaration	(change/add/delete) class the declaration/definition statements of the class $[x]$ (to modify $[x]$' $[properties]$)
Patch Pattern	Modify Method Declaration	(change/add/delete) the (declaration/definition) statements of the method $[x]$ (to modify $[x]$'s $[properties]$)
	Modify Method Invocation	(change/add/delete) the invocation of method $[m_{original}]$(, modifying it to $[m_{target}]$ by adjusting method $[properties]$))
	Modify Error Handling	(change/add/delete) (errors/exceptions) (handling code/approach to managing errors/exceptions within the program flow)
	Modify Sanity Check	(add/change/delete) a sanity check (on variable $[x, ...]$) to verify the security of a specific program state (by invoking m method/an assertion/...)
	Move Statements	move the position of the statements (in the context without modification)
	Others	no template: uncommon minor changes that cannot be categorized into any of the above types.

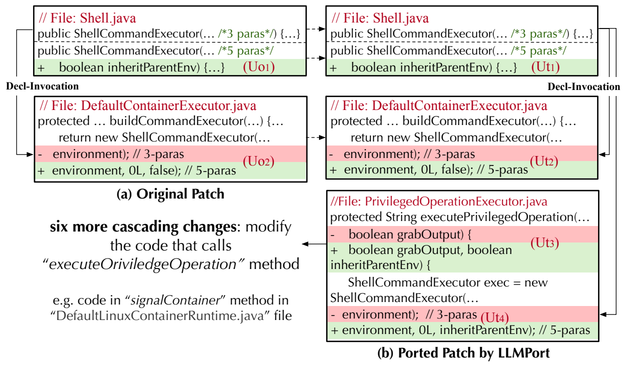
series of patch units with logical dependencies. Specifically, LLMPort performs a static analysis to identify dependencies between code elements in different patch units. We consider three types of dependencies: (i) structural relationship, including the inheritance between classes (*Inheritance*), and the override relationships between their methods (*Override*); (ii) control-flow dependencies, including the declaration and invocation of methods (*Decl-Invocation*), and the dependencies between the sanity check and its corresponding processing block (*Check-Handle*); (iii) data-flow dependencies, including the passing of variables during method calls (*Inter-process data dependencies*), and the definition and use of variables (*Def-Use*). For instance, consider U_{o1} and U_{o2} in Figure 3 (a). U_{o1} is a patch unit with the pattern "Modify Method Declaration" and U_{o2} involve "Modify Method Invocation". Since U_{o2} invokes the method declared in U_{o1}, these two units have a *Decl-Invocation* dependency. We should first port the method declaration unit and then the method invocation unit. Thus, LLMPort labels U_{o1} as the *dominator* unit of U_{o2}. We conduct similar methods to identify dominator units in other types of unit dependencies. For example, the variable declaration unit is a dominator of its usage and a super-class declaration unit is a dominator of its sub-classes.

B. Patch-related Code Context Alignment

After grouping the patch units and subtasks, LLMPort extracts the minimal code context necessary for the LLMs to analyze each subtask. This helps further focus the LLMs' attention. On one hand, LLMPort identifies the range of locations for the patch units in the target OSS version to narrow down the code content that LLMs need to analyze. On the other hand, LLMPort analyzes and summarizes the cross-version code differences that affect patch migration. Specifically, it focuses on two types of differences: (1) location and number of patch units, and (2) changes in code content.

As previously mentioned, patch units could depend on one another. To help LLMs account for earlier modifications during migrations, LLMPort first generates a subtask and ports the *dominant* unit. It treats the units as a tree and uses a Breadth-First Search strategy to process them in order, starting from the root unit, which is independent of others. We now describe how LLMPort generates a subtask for each patch unit.

1) Identifying Target Patch Unit Location: Given that LLMs can understand large code contexts [11], LLMPort selects a coarse-grained target region in V_{tar} for each patch unit


Decl-Invocation
Decl-Invocation
(a) Original Patch
six more cascading changes: modify the code that calls "executeOriviledgeOperation" method
e.g. code in "signalContainer" method in "DefaultLinuxContainerRuntime.java" file
(b) Ported Patch by LLMPort
(Uo1)
(Uo2)
(Ut1)
(Ut2)
(Ut3)
(Ut4)

```
// File: Shell.java
public ShellCommandExecutor(... /*3 paras*/) {...}
public ShellCommandExecutor(... /*5 paras*/
+   boolean inheritParentEnv) {...}
```

```
// File: DefaultContainerExecutor.java
protected ... buildCommandExecutor(...) {...
    return new ShellCommandExecutor(...
-   environment); // 3-paras
+  environment, 0L, false); // 5-paras
```

```
// File: Shell.java
public ShellCommandExecutor(... /*3 paras*/) {...}
public ShellCommandExecutor(... /*5 paras*/
+   boolean inheritParentEnv) {...}
```

```
// File: DefaultContainerExecutor.java
protected ... buildCommandExecutor(...) {...
    return new ShellCommandExecutor(...
-  environment); // 3-paras
+  environment, 0L, false); // 5-paras
```

```
//File: PrivilegedOperationExecutor.java
protected String executePrivilegedOperation(...
-   boolean grabOutput) {
+   boolean grabOutput, boolean
inheritParentEnv) {
    ShellCommandExecutor exec = new
ShellCommandExecutor(...
-  environment);  // 3-paras
+ environment, 0L, inheritParentEnv); // 5-paras
```

Figure 3: Patches for CVE-2016-3086 on hadoop

and delegates the precise insertion point to the LLM. For intra-method changes, LLMPort maps the modified method from V_{org} to V_{tar} and uses it as context for migration. For inter-method changes, it performs class-level mapping and provides the surrounding class-level code as context. To support this, LLMPort aligns each patch unit to its corresponding method or class in V_{tar} using an alignment strategy based on [13].

However, differences in vulnerability scope and evolving code dependencies across versions often require an N-to-N mapping between patch units and their target locations. LLMPort address this by constructing a dependency graph among patch units. This graph guides the generation of subtasks, enabling more complete and reliable patch porting.

N-to-N Patch Unit Mapping. First, we illustrate how LLMPort handles N-to-N mappings caused by changes in vulnerability scope. Specifically, LLMPort identifies two decode methods in V_{tar} and generates two subtasks for unit U_{o3} at V_{org} (Figure 1), targeting the decode methods in both RaftSyncMessageSerializer and RaftSnapshotSerializer classes (i.e., one-to-two mapping). To ensure completeness, LLMPort examines the patch dependency graph, identifies units related to U_{o3} (e.g., U_{o1}, U_{o2}), and generates supplement subtasks to port these dependent units.

Second, we detail how LLMPort addresses another N-to-N mapping issue arising from *Decl-Invocation* dependencies as an example. The core idea is that it would identify all the related code units in V_{tar} based on the dependency. Then, for each of them, it would find a similar unit in V_{org} if possible as a reference to guide the LLM. The approaches for other dependencies, i.e., *overrides*, *Def-Use*, etc., are similar.

Take the patch for vulnerability CVE-2016-3086 in Figure 3 as an example. U_{o2} changes the invoked constructor of ShellCommandExecutor class from three to five parameters. Thus, considering the dependency of *Decl-Invocation*, all the invocations of this three-parameter constructor (U_{t2} and U_{t4} in Figure 3(b)) should be identified and adjusted in V_{tar}, which could be very different with those (e.g., U_{o2}) in V_{org}.

Then, for adjusting U_{t2} and U_{t4}, LLMPORT would try to identify a similar code unit for each of them in P_{org} and then use it as a reference to guide the LLM. For example, it will first extract the code lines of U_{t2} and code (at most twenty lines) surrounding it to construct a code segment. Then, it extracts the code segments for U_{o2} in a similar way. Since the code segment carries the code context for each unit, LLMPORT further compares the code similarity of these two segments to check if U_{t2} could map to U_{o2}. If the similarity score exceeds the threshold (0.7), LLMPORT will use the patch code of U_{o2} in V_{org} to guide the LLM for adjusting U_{t2} in V_{tar}. Otherwise, it would let the LLM decide how to adjust U_{t2}.

In addition, LLMPORT considers continuous mapping variations. If the porting process results in a change in the declaration of another method, e.g., the adjustment of U_{t4} introduces the changes of `executePrivilegedOperation()` (U_{t3}) in Figure 3 (b), LLMPORT searches for the code that invokes this method and constructs new patch porting subtasks.

2) Identifying Code Element Difference: LLMs excel at understanding-based code generation, eliminating the need for accurately identifying code adjustment strategies, which can be particularly challenging. In our experiments (§ V-C), the state-of-the-art porting tools FixMoph and TSBPORT achieved accuracies of only 22.64% and 33.96%, respectively, in cases requiring code content adjustments. Thus, we prompt LLMs to make specific code adjustments based on changes in the properties of code elements which is feasible to accurately identify instead. If no changes, we only use LLMs to pinpoint the exact modification locations.

LLMPORT iterates over each code element in the official patch unit and maps it to V_{tar}. It locates corresponding elements for invoked methods and instantiated or extended classes, and matches defined or used variables using GumTree [15]. It then parses and records the changed properties of mapped code elements across versions. For example, when identifying the vulnerable `decode` method in the Raft snapshot module in V_{tar}, which exhibits similar vulnerability patterns, LLMPORT detects a change in the return type to `RaftSnapshot`. It guides the LLM in analyzing the differences and adapts the code by adding two `RaftSnapshot` class names in `PERMITS` at Line 4-5 in Figure 1 (b).

C. Prompting LLMs for Patch Porting

Dependency-aware patch porting. Overall, LLMPORT automatically creates a prompt for each subtask from the root and directs LLMs to generate patch units for V_{tar} in sequence. To facilitate this process, LLMPORT employs a dependency-aware mechanism that propagates cross-version patch changes across interdependent units (subtasks) through control-flow, data-flow, and structural links. This mechanism enables coordinated multi-location refinement, surpassing the limitations of previous locally scoped adjustment strategies, such as TSBPORT. For instance, as shown in Figure 3, LLMPORT ports the dominant patch U_{o1} before U_{o2}. This order helps LLMs to generate appropriate modifications for U_{t4} based on the earlier changes in U_{t1}, despite the absence of official patch unit references. It then adjusts cascading changes introduced by U_{t4}, e.g., U_{t3}.

The prompt that LLMPORT uses for each subtask consists of three main segments. (1) **System role.** LLMPORT first sets the system role for the LLM, instructing it to port a patch from V_{org} to V_{tar} while respecting language-specific characteristics. (2) **Subtask description.** Then, LLMPORT generates a detailed task description for each porting task. This includes a summary of changes in patch-related code elements from V_{org} to V_{tar}, along with code adjustment suggestions to guide the LLM's focus. LLMPORT analyzes four types of patch-related context differences between versions: (i) patch location, (ii) code elements, (iii) properties of code elements, and (iv) original code statements related to modifications. To maintain a global perspective during subtask migration, LLMPORT also extracts cross-version patch code differences from directly neighboring subtasks that have been migrated along with the current subtask. For example, since subtask-3 is related to subtask-2 in Figure 1, LLMPORT also focuses on the difference related to subtask-3, i.e., the changes to the code that handles deserialized objects after they have passed the sanity checks. Finally, LLMPORT instructs the LLM to generate output in JSON format, enabling downstream tooling to locate and apply the required edits consistently and precisely. (3) **Patch-related code context.** LLMPORT provides the patch-related context for the current subtask. This includes the official patch unit as a reference, the vulnerable code fragment in V_{tar} that needs to be modified based on context alignment, and the directly related ported patch unit. Specifically, for each official patch unit, LLMPORT extracts the modified lines within a range of twenty surrounding lines to preserve the local context. In V_{tar}, LLMPORT adaptively slices the context into two categories: (i) for intra-method regions, it includes the method declaration and body; (ii) for non-method regions, it encompasses class declarations, fields, and static blocks, excluding methods. This approach ensures contextual completeness while minimizing the inclusion of irrelevant code.

IV. LLM-DRIVEN PATCH CORRECTION

After porting all subtasks, LLMPORT generates the complete patch (P_{tar}) and applies it to produce the fixed target program. However, the resulting patch may contain errors that render it unusable. LLMPORT leverages multiple auditing tools and integrates static analysis with LLMs to diagnose root causes from error reports. It then synthesizes fine-grained prompts to guide the LLM in correcting the faulty parts.

A. Prompting LLMs for Patch Correction

The prompt that LLMPort uses to fine-tune the incorrect patch consists of three main components. (1) **System role.** LLMPort sets the LLM's role to fix error patch code while respecting language-specific characteristics. (2) **Error Feedback.** This component contains two parts: (i) an error report, which includes error messages, cause analysis, and adjustment suggestions; and (ii) the faulty code and its surrounding context for correction. We discuss how to parse and generate this component at § IV-B. (3) **Audit summary.** This component includes two types of summaries. First, LLMPort summarizes the adjustments history for the current error subtask. For example, if LLMPort makes repeated attempts to fix the same error subtask, it summarizes previous adjustments to avoid repeated mistakes. Second, LLMPort summarizes the patch-related context differences associated with the current subtask. This helps guide the LLM in making accurate, continuous adjustments.

B. Auditing Errors and Generating Feedback

LLMPort first invokes a compiler to verify the syntactic correctness, structural completeness, and build compatibility of P_{tar}. It mainly supports six kinds of fine-grained error reasons (E#1-E#6), which account for 74.89% of the cases in our evaluation, as detailed below. However, compilation errors are often coarse-grained and insufficient to guide precise LLM edits. To address this, LLMPort extracts error-related code elements and applies static analysis to localize and classify fine-grained root causes. In addition, since compilers cannot guarantee functional correctness, LLMPort further identifies and executes targeted test suites, detailed in E#7, to ensure the patch's effectiveness and behavioral compatibility.

E#1: Missed code dependencies. LLMs may inadvertently ignore some code dependencies during patch porting, resulting in compilation errors, e.g., "cannot resolve symbol" or "class *c* must either be declared abstract or implement abstract method". To identify this problem, LLMPort will check if the code context required by the erroneous code lines is missing in V_{tar}. Specifically, it utilizes static analysis to examine every code element in the code lines where the compilation error occurs and identifies missed context typically arising from dependencies between code elements, including *Def-Use*, *Decl-Invocation*, and *Override*. LLMPort does not directly introduce missing dependencies as new variables, classes, or methods. Instead, it first searches V_{org} for the original definitions and uses cross-version mapping to identify semantically equivalent dependencies in V_{tar}. If such dependencies are found, LLMPort guides the LLM to adapt the imports accordingly. If no equivalent dependencies are found, new definitions are introduced based on V_{org}. In cases where the required dependencies are missing in both versions, LLMPort instructs the LLM to synthesize new definitions based on the usage context.

E#2: Incorrect patch location. As discussed, LLMPort relies on LLMs to determine the specific locations of patch units. However, LLMs may sometimes identify incorrect patch locations. To address this, LLMPort analyzes prior porting traces and aligned inter-version contexts before applying each ported unit. This allows LLMPort to detect and prompt LLMs to revise invalid modification insertion points. By doing so, LLMPort prevents (1) the overwriting of non-vulnerable code and (2) redundant edits at the same location that might overlook other vulnerable code in similar contexts. Additionally, LLMPort identifies residual invalid patch locations after applying the complete P_{tar} by capturing compilation or runtime failures. One case is that, LLMs may put the declaration of a variable after its usage, resulting in a compilation error of "cannot resolve symbol". To identify this problem, LLMPort will check the declaration of variables in the code that failed to compile. If one variable declaration appears after its usage, the location of the corresponding patch unit may be incorrect. Then, LLMPort will prompt LLMs to re-locate the patch units related to the definition and usage of this variable.

E#3: Type and interface mismatches. Compiler messages such as "incompatible types: ... cannot be converted to ..." indicate type mismatches or interface conformance violations. For such errors, LLMPort traces the relevant function or method signatures, extracts mismatched arguments or return expressions along with their expected types, and prompts the LLM to revise the code accordingly.

E#4: Errors in handling code element properties. Compilation errors like "field *f* has private access in class *c*" indicate the LLMs do not well handle code element properties. LLMPort identifies the code element in the error report. Then, it extracts surrounding scope declarations (e.g., the fields' declaration in class *c*).

E#5: Code elements mapping error. In § III-C, LLMPort extracts only one method or class with the highest similarity for the patch porting task. However, ensuring accuracy becomes challenging when the code has changed significantly between versions, especially for more complex code refactorings that are beyond the scope of what we have considered in § III-B1. To address this, LLMPort checks for errors that may arise from its inability to accurately map code elements. It prompts LLMs to consider alternative methods or classes with the top ten similarity ranking. The LLMs are then prompted to reanalyze these alternatives and select the most appropriate ones based on the context.

E#6: Syntactic and other compilation errors. For other compilation errors detectable by the compiler, such as syntax errors and missing import packages, etc., LLMPort extracts the compilation reports and relevant code segments for the LLMs to analyze and fix independently. LLMs can effectively resolve most common errors that do not involve code logic, e.g., "',' or ')' expected". The compiler also helps identify complex issues such as Java version incompatibilities, where differences in supported syntax or APIs produce compile-time errors that often indicate deeper semantic mismatches. LLMPort leverages these diagnostics to guide LLMs in adapting code to the target Java version, enabling flexible corrections beyond traditional static analysis.

E#7: Failures in test suites or other errors. LLMPort extracts relevant test suites whose control flow, determined by the Call Graph, reaches the methods modified by the patch. For functional testing, we use the existing relevant test suites in V_{tar}. For security testing, we port test suites identified from P_{org} to V_{tar}. Then, LLMPORT execute these test suites. For failed tests, LLMPORT first extracts the error logs and identifies the error code along with the associated subtask(s). Then, it prompts- the LLM in refining the patch to resolve conflicts between the patch modifications and business functionality, or to further enhance the security fix.

V. EVALUATION

We assess LLMPORT on real-world security patch porting.

Verification process. For each generated patch, we verify[4] whether it can successfully repair the same vulnerability on V_{tar} regarding ground-truth (GT) criteria (i.e., official patches), aligning with existing works [25], [35]. Specifically, three authors discuss until consensus with the following steps: ❶ Check that the generated patch unit matches the GT, i.e., there are no missing or redundant modifications; ❷ Check that the patch location in the context is consistent or equivalent to the GT; ❸ Verify the generated patch is free of syntax errors and is semantically and logically equivalent to GT. In the experiments in § V-B and § V-C, we employ the same validation process as TSBPORT for porting tasks without GT.

A. Comparison with out-of-the-box LLMs

In this section, we apply LLMPORT to four representative LLMs and evaluate its effectiveness. Specifically, we chose three commercial LLMs, i.e., GPT-4 (-1106-preview) [9], GPT-3.5 (-turbo) [7], qwen (-max) [10], and an open-source LLM, i.e., gemma (gemma-7b-it) [30].

Datasets. Tan et al. [29] found that developers take an average of 233 days to manually port patches for Java projects, highlighting the need for automated patch porting tools. However, to the best of our knowledge, no such tools currently exist for Java projects. To address this research gap, we primarily apply LLMPORT for Java programs and evaluate its effectiveness.

Due to the absence of large-scale open-source Java patch porting datasets, we constructed a benchmark, that contains 1,992 patch file pairs, and 707 patch pairs. In the benchmark, each patch has 2.06 patch files (except test files) and 111.38 lines of code modifications on average, more complex than the 5.36 lines in FixMorph's.

Overall Results. Table III shows the overall results, which presents the percentage of successfully ported patches using LLMPORT combined with different LLMs. Among them, LLMPORT-GPT-4 successfully ported 1,831 out of 1,992 patch file pairs, resulting in an overall success rate of 91.92%. Specifically, LLMPORT-GPT-4 obtained a 96.20% success rate when porting the Type-I patches. For complex porting tasks, which require code structure or even logic adjustments, it still achieved success rates of 90.67% for Type-II, 91.36% for Type-III, and 73.75% for Type-IV cases.

As aforementioned, Table I shows the results of patch porting by directly utilizing these four out-of-the-box LLMs without LLMPORT. A comparison of the experimental results in Table III and Table I shows a significant improvement in patch porting success rates after applying LLMPORT, demonstrating its effectiveness. For instance, in the most challenging Type IV cases, LLMPORT-GPT-4 increased the success rate from 8.26% (in Table I) to 73.75% (in Table III), representing a substantial performance improvement of 64.49%. Additionally, all LLMs enhanced with LLMPORT show significant improvements, demonstrating the versatility of LLMPORT. Specifically, LLMPORT achieved improvements of 47.49%, 58.41%, and 27.32% on the other three LLMs, respectively, which originally did not perform as well as GPT-4.

Ablation Study. We conducted an ablation study comparing direct LLM usage, LLMPORT-*LLM*-NoCorr (LLMPort without self-correction), and the full LLMPORT. The results emphasize the critical role of two key modules in the LLMPORT.

Specifically, though the four LLMPORT-*LLM*-NoCorr approaches only leverage the subtask-based porting module, they achieve significantly higher effectiveness compared to the direct use of LLMs. Our further analysis reveals that, the subtask-based porting approach reduces the code context and complexity LLMs need to analyze for each porting, thus reducing missed code changes (E#1) and improving patch localization accuracy (E#3), which are very common mistakes.

Additionally, the full implementation of LLMPORT has the highest success rates consistently, with more pronounced improvements in more complex porting tasks (i.e., Type-III–IV). Complex patch porting tasks often involve significant contextual differences and numerous code modifications, making it challenging for LLMs to generate valid patches in a single attempt. While the subtask-based patch migration module improves LLMs migration success rates in complex cases, such as increasing GPT-4's success rates in Type-III and Type-IV cases from 18.07% and 8.26% to 28.92% and 36.28%, respectively, most complex migrations remain unmanageable. Therefore, LLMPort's self-correction system is necessary to port complex patches, increasing the efficiency of GPT-4 on Type-III–IV cases to 91.92% and 73.75%, respectively.

Failure Cases. We examine LLMPORT-GPT-4, the best-performed LLM, to understand why LLMPORT would fail to specific patch porting tasks. Table IV shows the number of incorrect patch files ported by LLMPORT-GPT-4-NoCorr and LLMPORT-GPT-4. We manually analyze these incorrect files and classify them based on the reasons discussed in § IV. It is important to note that one failed porting case may have multiple error reasons. Overall, with the self-correction module, LLMPORT can fix most (82.19%) of the errors generated by the subtask-based patch porting module, indicating its effectiveness. However, 221 errors still exist. Our manual analysis identifies that errors occurring during the patch context alignment stage include 17 instances of incorrect code element matches (E#5) and 21 misidentified

[4]We first use an automated tool to filter the patches generated by the LLMs that are completely consistent with the GT, and then manually recheck the validity of inconsistent patches.

Table III: The results of combining LLMPORT with four out-of-the-box LLMs in the benchmark.

Approach	Patch File Pair					Patch Pair	Time Cost		Average Money Cost		
	Type-I	Type-II	Type-III	Type-IV	Total		Stage-I	Stage-II	Stage-I	Stage-II	Total
LLMPORT-GPT-4	1,369 (96.20%)	136 (90.67%)	75 (90.36%)	251 (73.75%)	1,831 (91.92%)	591 (83.59%)	9m25s	6m11s	\$0.17	\$0.95	\$1.12
LLMPORT-GPT-3.5	1175 (82.75%)	106 (70.67%)	66 (79.51%)	181 (53.39%)	1,528 (76.71%)	513 (72.56%)	8m34s	7m51s	\$0.02	\$0.24	\$0.26
LLMPORT-qwen	1,276 (89.86%)	123(82.00%)	70 (84.34%)	211 (62.24%)	1,680 (84.34%)	553 (78.22%)	16m27s	10m19s	\$0.10	\$0.73	\$0.83
LLMPORT-gemma	780 (60.56%)	58 (42.03%)	32 (41.03%)	93 (28.79%)	963 (52.71%)	421 (50.24%)	13m21s	9m11s	-	-	-

Table IV: Distribution of Error Causes in Cases Before and After Patch Correction by LLMPort-GPT-4

	E#1	E#2	E#3	E#4	E#5	E#6	E#7
LLMPORT-GPT-4NoCorr	37	478	38	101	78	197	312
LLMPORT-GPT-4	0	82	0	26	17	11	85

patch locations (E#2). A significant proportion of errors (i.e., 183 cases in E#2, E#4, E#6, E#7) originate during the initial patch generation phase, with no new errors introduced during the correction phase. Upon further analysis of the error causes, we find that these inaccuracies are primarily due to the substantial differences between the original and target versions of the code. These differences can affect the accuracy of the method and class-level similarity mapping of LLMPORT, leading to errors such as E#2 and E#5. Additionally, prior updates to V_{org} may have introduced some security features, and these are not explicitly connected to the current patch code through data or control flow dependencies, making them difficult to identify and integrate into V_{tar}. Thus, LLMPORT may miss porting some code, leading to errors such as E#7.

Performance. We evaluate LLMPORT's performance with different LLMs, detailing its time and money cost at each stage in Table III. For example, LLMPORT-GPT-4 takes 9 minutes and 25 seconds in stage I, and 6 minutes and 11 seconds in stage II, and costs \$1.12 on average for porting a patch. The performance breakdown shows that stage I is the most time-consuming. This is primary because LLMPORT needs to thoroughly analyze the repair behavior of the official patch and fully assess the patch-related code structure and business logic differences across versions. This analysis is crucial for generating high-quality subtasks that guide the subsequent step-by-step patch generation and fine-tuning, especially for large projects and complex patches. Furthermore, the patch correction stage incurs higher costs due to the need for LLMPORT to interact with multiple external auditing modules and iteratively adjust the generated patches. For complex cases, this process may involve over ten iterations of adjustments.

B. Generalizability Evaluation

In this section, we evaluate LLMPORT's generalization in porting patches for vulnerabilities across various projects, all of which fall outside the LLMs' training data. Due to undisclosed cut-off dates for qwen and gemma, we evaluate LLMPORT using GPT-3.5 and GPT-4 (April 2023).

Datasets. First, we randomly selected 50 vulnerabilities affecting popular Java apps since April 2023 outside the test dataset. Then, we collected 69 pairs of patches (i.e., $P_{org}s$ and other vulnerable V_{tar}s) as the test dataset, of which 36.23% required code content changes when porting across versions.

Overall Results. LLMPORT-GPT-4 successfully ported 56 out of 69 patches, achieving an overall success rate of 81.16%. LLMPORT-GPT-3.5, also managed to port 51 patches, having a 73.91% overall success rate. The experimental results indicate that LLMPORT performs well on other vulnerabilities and OSS projects outside LLMs' training data, and also prove the generalization of domain knowledge in LLMPORT.

Accepted and merged patches. By employing LLMPORT to port these patches, we observed that it attempted to fix additional vulnerable code snippets that were overlooked by the original patches. As a result, we discovered five 0-day vulnerabilities, four of which have already been assigned CVE IDs. To assess the practical effectiveness of LLMPORT-generated patches and assist developers in project maintenance, we submitted these patches to the respective branches. Currently, all of them have been merged by the developers of Apache Seata [1] (25.3k stars), Undertow [2] (3.6k stars), Solon [4] (2.3k stars), and mpxj [3] (246 stars). An example is illustrated in Figure 1 as our motivation.

C. Portability Evaluation

We also evaluated LLMPORT on C patch porting tasks using FixMorph's dataset and compared it against two other end-to-end tools: FixMorph and TSBPORT. We re-executed TSBPORT and FixMorph using their official Docker environments and instructions. Note that our patch porting classification differs from FixMorph, so we re-fined the dataset by merging Type-I, Type-II from FixMorph into Type-I and splitting Type-VI into Type-I [5] –II and IV.

Overall Results. Table V summarizes the overall results. Compared with TSBPORT and FixMorph, LLMPORT achieves the highest porting success rate in all four types of patches. For example, LLMPORT-GPT-4 improves the success rate by 40.57%, 58.33%, 61.53%, and 69.77% when porting Type-I, Type-II, Type-III, and Type-IV patches compared with TSBPORT, respectively. LLMPORT also performs better when porting C language patches compared with porting Java patches. This improvement is attributed to the simplicity of the patch dataset in comparison to our Java benchmark.

VI. DISCUSSION

Support for multi-file patches. Our analysis shows that 52.24% of 1,583 patches involved multiple file modifications. These multi-file patches often face significant cross-version

[5] We identified and corrected eight patch pairs from FixMorph's dataset mistakenly labeled as Type-IV despite having identical modified code.

Table V: Success ported security patches comparison among LLMPort, TSBPORT, FixMorph in C (FixMorph dataset)

Porting Type	Patch Pairs	LLMPort -GPT-4	LLMPort -GPT-3.5	TSBPORT	FixMorph
Type-I	244	243 (99.59%)	239 (97.95%)	144 (59.02%)	210 (86.07%)
Type-II	24	23 (95.83%)	23 (95.83%)	9 (37.50%)	16 (66.67%)
Type-III	39	35 (89.74%)	32 (82.05%)	11 (28.21%)	12 (30.77%)
Type-IV	43	34 (79.07%)	31 (72.09%)	4 (9.30%)	8 (18.60%)
Total	350	335 (95.71%)	325 (92.86%)	168 (48.00%)	246 (70.29%)

code differences and complex dependencies, making them particularly challenging and time-consuming for developers to migrate manually. However, existing tools (e.g., [21], [25], [35]), typically overlook this common issue, supporting only single file patch porting. Compared with them, LLMPort's subtask-based decomposition mechanism and patch iterative tuning system based on LLM's self-correction capability benefit it in overcoming this problem and support multi-file patches.

Reliance on original patch. Our work shows that, reusing patches that humans already tested is more effective than having AI create new fixes from scratch. As a comparasion, prior work [23] shows only 5.01% AI-made fixes actually worked without causing new problems. When we tested GPT-4, it only fixed six bugs – far fewer than human-tested fixes could solve.

VII. Related Work

Patch Porting. Several studies on patch backporting have been proposed in recent years. In general, there are two primary approaches. FixMorph [25] develops syntax-based methods with pre-defined rules to backport the patch to the target version. However, this type of solution is not applicable in the case of huge syntactic structure differences across versions, which is a common scenario in OSS. Some work [26] [28] [35] implement semantic-based approaches. Their main idea is to match the patch to unpatched code depending on code semantic knowledge. PatchWeaver [26] uses exploits to achieve this goal and implements a concolic execution technique to backport the patch. However, exploits are often missing in the real world, and concolic execution has great computational overhead; this approach lacks scalability. SKYPORT [28] and TSBPORT [35] rely on predefined code patterns and rules, limiting their scalability. Compared to these works, LLMs have a cross-domain corpus and can handle large contexts, demonstrating strong generalization in analyzing complex code and generating comprehension-based code without relying on predefined rules. PPatHF [21] employs instruction-tuned LLMs for automated patch porting in hard forks but cannot port patches with numerous modifications or handle dependencies across different code segments within a single patch. Compared to this work, LLMPort can handle larger-scale, multi-file patches through its task decomposition and self-correction mechanisms.

Patch generation. Another thread of related work is patch generation. PraPR [16] uses bytecode mutation to fix real-world bugs. Par [18] utilizes human-written patches to develop a learning-based patch generation method. TBar [19] further summarizes fix patterns from literature to improve generation accuracy. However, these studies have similar drawbacks: they require predefined templates and test cases, limiting scalability. To overcome this, rule-based patch generation methods were introduced. Elixir [24] combines heuristic rules with machine learning, CapGen [32] uses AST ranking rules, and Arja [36] employs genetic mutation rules. While these methods reduce manual effort, they suffer from high false positives and struggle with corner cases. Additionally, deep learning-based approaches such as VRepair [12], which uses transfer learning, and Mashhadi et al. [20], which fine-tunes CodeBert for Java bugs, show promise. However, their performance is limited by the large number of parameters across languages and bug types. With the rise of LLMs, ChatRepair [33] uses ChatGPT for recursive bug fixing. However, [23] shows that LLMs are not suitable for direct patch generation, with only a 5.01% reliable patch generation success rate.

VIII. Conclusion

This paper introduces LLMPort, a novel and portability LLM-driven approach for cross-version security patch porting. It divides a complex patch porting task into distinct subtasks, extracting minimal patch-related code context and constructing task-specific prompts to guide LLMs in generating corresponding patch code. Additionally, LLMPort includes a progressive self-correction system to assess correctness and repair generated error patches automatically. We evaluate LLMPort's performance for porting Java and C language patches, demonstrating its effectiveness and portability.

Acknowledgement

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172104, 62172105, 62202106, 62302101, 62102091, 62472096, 62402114, 62402116). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012).

References

[1] 2024. github apache/incubator-seata repository. https://github.com/apache/incubator-seata.
[2] 2024. github io/undertow repository. https://github.com/undertow-io/undertow.
[3] 2024. github joniles/mpxj repository. https://github.com/joniles/mpxj.
[4] 2024. github opensolon/solon repository. https://github.com/opensolon/solon.
[5] diff tool. https://manpages.ubuntu.com/manpages/trusty/man1/diff.1posix.html.
[6] GitHub - tree-sitter/tree-sitter: An incremental parsing system for programming tools. https://github.com/tree-sitter/tree-sitter.
[7] Openai. 2022. GPT-3.5 Model Registry. https://platform.openai.com/docs/model-index-for-researchers/models-referred-to-as-gpt-3-5.
[8] May 2023's most wanted malware: New version of Guloader delivers encrypted Cloud-Based Payloads - Check point software, 6 2023.

[9] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

[10] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, et al. Qwen technical report. arXiv preprint arXiv:2309.16609, 2023.

[11] 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.

[12] 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.

[13] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. Facilitating vulnerability assessment through poc migration. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 3300–3317, 2021.

[14] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. IEEE Security & Privacy, 20(2):96–100, 2022.

[15] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pages 313–324, 2014.

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

[17] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. arXiv preprint arXiv:2210.02406, 2022.

[18] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In 2013 35th international conference on software engineering (ICSE), pages 802–811. IEEE, 2013.

[19] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, pages 31–42, 2019.

[20] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pages 505–509. IEEE, 2021.

[21] 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.

[22] Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl Barr, and Sergey Mechtaev. The fact selection problem in llm-based program repair. arXiv preprint arXiv:2404.05520, 2024.

[23] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In 2023 IEEE Symposium on Security and Privacy (SP), pages 2339–2356. IEEE, 2023.

[24] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented program repair. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 648–659. IEEE, 2017.

[25] 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.

[26] 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.

[27] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In International Conference on Machine Learning, pages 31210–31227. PMLR, 2023.

[28] 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.

[29] 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, pages 767–777, 2022.

[30] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. arXiv preprint arXiv:2403.08295, 2024.

[31] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 149–160. IEEE, 2021.

[32] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In Proceedings of the 40th international conference on software engineering, pages 1–11, 2018.

[33] 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.

[34] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In 29th USENIX Security Symposium (USENIX Security 20), pages 2397–2414, 2020.

[35] 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.

[36] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. IEEE Transactions on software engineering, 46(10):1040–1067, 2018.

[37] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. Patchscope: Memory object centric patch diffing. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 149–165, 2020.