MYSTIQUE: Automated Vulnerability Patch Porting with Semantic and Syntactic-Enhanced LLM

SUSHENG WU*, Fudan University, China
RUISI WANG*, Fudan University, China
YIHENG CAO, Fudan University, China
BIHUAN CHEN†, Fudan University, China
ZHUOTONG ZHOU, Fudan University, China
YIHENG HUANG, Fudan University, China
JUNPENG ZHAO, Fudan University, China
XIN PENG, Fudan University, China

Branching repositories facilitates efficient software development but can also inadvertently propagate vulnerabilities. When an original branch is patched, other unfixed branches remain vulnerable unless the patch is successfully ported. However, due to inherent discrepancies between branches, many patches cannot be directly applied and require manual intervention, which is time-consuming and leads to delays in patch porting, increasing vulnerability risks. Existing automated patch porting approaches are prone to errors, as they often overlook essential semantic and syntactic context of vulnerability and fail to detect or refine faulty patches.

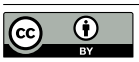
We propose MYSTIQUE, a novel LLM-based approach to address these limitations. MYSTIQUE first slices the semantic-related statements linked to the vulnerability while ensuring syntactic correctness, allowing it to extract the signatures for both the original patched function and the target vulnerable function. MYSTIQUE then utilizes a fine-tuned LLM to generate a fixed function, which is further iteratively checked and refined to ensure successful porting. Our evaluation shows that MYSTIQUE achieved a success rate of 0.954 at function level and of 0.924 at CVE level, outperforming state-of-the-art approaches by at least 13.2% at function level and 12.3% at CVE level. Our evaluation also demonstrates MYSTIQUE's superior generality across various projects, bugs, and programming languages. MYSTIQUE successfully ported patches for 34 real-world vulnerable branches.

CCS Concepts: • **Security and privacy**; • **Human-centered computing** → **Open source software**;

Additional Key Words and Phrases: Patch Porting, Large Language Model, Open Source Software

*Both authors contributed equally to this work.
†Bihuan Chen is the corresponding author.

Authors' Contact Information: Susheng Wu, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, scwu24@m.fudan.edu.cn; Ruisi Wang, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, rswang23@m.fudan.edu.cn; Yiheng Cao, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, caoyh23@m.fudan.edu.cn; Bihuan Chen, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, bhchen@fudan.edu.cn; Zhuotong Zhou, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, zhouzt23@m.fudan.edu.cn; Yiheng Huang, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, yihenghuang23@m.fudan.edu.cn; JunPeng Zhao, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, 24210240422@m.fudan.edu.cn; Xin Peng, Affiliated with School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, pengxin@fudan.edu.cn.



ACM 2994-970X/2025/7-ARTFSE007
https://doi.org/10.1145/3715718

ACM Reference Format:
Susheng Wu, Ruisi Wang, Yiheng Cao, Bihuan Chen, Zhuotong Zhou, Yiheng Huang, JunPeng Zhao, and Xin Peng. 2025. Mystique: Automated Vulnerability Patch Porting with Semantic and Syntactic-Enhanced LLM. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE007 (July 2025), 23 pages. https://doi.org/10.1145/3715718

1 Introduction

In modern software development, branching within a single repository provides an efficient way to manage and organize feature development [23, 36, 38]. However, this practice can inadvertently propagate vulnerabilities. When a branch is created from another, it may inherit existing vulnerabilities from the parent branch. Even if the original branch is later fixed with a patch, the target unfixed branches within the repository remain vulnerable unless the patch is consistently applied across them [20, 42]. To mitigate these risks, a straightforward and cost-effective approach is to port security patches from the original branch to target branches. However, due to the inherent discrepancies between branches [46], many patches cannot be directly applied, resulting in delays in patch porting and increasing vulnerability exposure [20, 41].

Existing Approaches. Various approaches have been proposed to automatically port patches for mitigating this problem. Pattern-based approaches, such as FixMorph [41] and TSBPORT [62], rely on predefined patch patterns to automatically port patches within local code hunks. FixMorph [41] primarily utilizes syntactic information to port patches. However, its heavy reliance on syntactic patterns often leads to the neglect of semantic aspects within the local hunks. Additionally, the patch generation process in FixMorph depends on compilation by Clang/LLVM, which is time-consuming and lacks generalizability across different repositories. To address these limitations, TSBPORT [62] introduces a range of semantic hunk types to enhance the porting process but still suffers from significant limitations, i.e., if any modified hunk falls outside the predefined patterns, the entire patch may fail, and the lack of function-level semantic information can result in the introduction of unexpected patch porting errors.

LLM-based approach, exemplified by PPatHF [35], leverages a fine-tuned large language model (LLM) to automate the porting of patches across forked projects at function level. This approach benefits from LLM's ability to comprehend the discrepancies of a function between the original and target branches. PPatHF attempts to simplify the patching process by removing compound blocks with no modified lines, thereby focusing on core changes. However, this approach can inadvertently strip away crucial syntactic and semantic information related to the vulnerability, potentially resulting in patches that remain vulnerable. Moreover, PPatHF relies heavily on intra-repository knowledge for LLM fine-tuning, which can lead to misalignments with the actual porting task, thereby reducing its effectiveness across different projects or languages. Additionally, LLM may generate incomplete or incorrect patches, as PPatHF often overlooks the diverse reasons for LLM porting failure, which hinders further patch adaption and leads to unsuccessful patching.

Besides, there are some approaches designed for specialized vulnerability. For example, SkyPort [43] ports patches to earlier versions by matching portable constraints between modified lines and sink functions for PHP injection vulnerabilities. However, this approach requires manual curation of sink functions, limiting its applicability to certain types of vulnerabilities. PatchWeave [42] utilizes the exploitable test cases to guide patch porting. However, the vulnerability may lack exploitable test cases, limiting the practical effectiveness.

Challenges. We summarize two main challenges of the existing approaches. **(C1)** Pattern-based approaches work at the hunk level, often missing vulnerability-relevant semantic information outside the modified code segment, and LLM-based approach, though working at a broader function level, can still overlook vulnerability relevant statements and include irrelevant ones. The underlying problem is that both approaches struggle to effectively balance porting granularity and capturing

relevant context. **(C2)** Pattern-based approaches struggle with unexpected code variations, while LLM-based approaches often rely too heavily on generated output without thorough verification. This leads to unresolved porting failures and incomplete patches, as neither approach has a robust mechanism for detecting and refining these issues.

Our Approach. We propose a novel approach, Mystique, for automatically porting patches across different branches at function level. To address **C1**, Mystique extracts vulnerability-relevant signatures of the patch function (including pre-patch and post-patch function) from the original branch and then identifies corresponding signature of the vulnerable function from target branch. It then uses a fine-tuned LLM to port the patch. Specifically, Mystique slices control-flow and data-dependency related statements with the changed statements, and then further ensures the syntactic completeness for these sliced statements leveraging Abstract Syntax Tree (AST). Subsequently, Mystique generates the target fixed function using an LLM fine-tuned on the same task.

To address **C2**, Mystique first checks whether the vulnerable function is properly fixed, whether branch discrepancy is preserved, and whether LLM adheres to the prompt's constraints, applying heuristic rules to address these anomalies. Mystique then generates prompts with detailed suggestions based on these anomalies to guide LLM in refining the patches. This iterative checking-refining process ensures continuous improvement and increases the success rate of patch porting.

Evaluation. We assess Mystique's effectiveness by comparing it against two state-of-the-art pattern-based approaches, one leading LLM-based approach, and four advanced LLMs across 694 CVEs involving 1,359 vulnerable functions. The results demonstrate that Mystique successfully ports 1,297 (0.954) functions and 641 (0.924) CVEs, significantly outperforming the state-of-the-art approaches by at least 13.2% and 12.3%, respectively. Additionally, we conduct an ablation study and parameter sensitivity analysis to evaluate the contribution of each critical module and parameter in Mystique. Furthermore, we comprehensively evaluate the generality of Mystique across different projects, bugs, and programming languages (e.g., Java). Mystique exhibits superior generality, outperforming the state-of-the-art approaches by at least 112.5%, 18.6%, and 15.4% at function level in the three generality tasks. To assess Mystique's practical usefulness, we apply it to successfully port 34 real-world vulnerable branches, with 29 successfully merged.

Contribution. This work makes the following contributions.

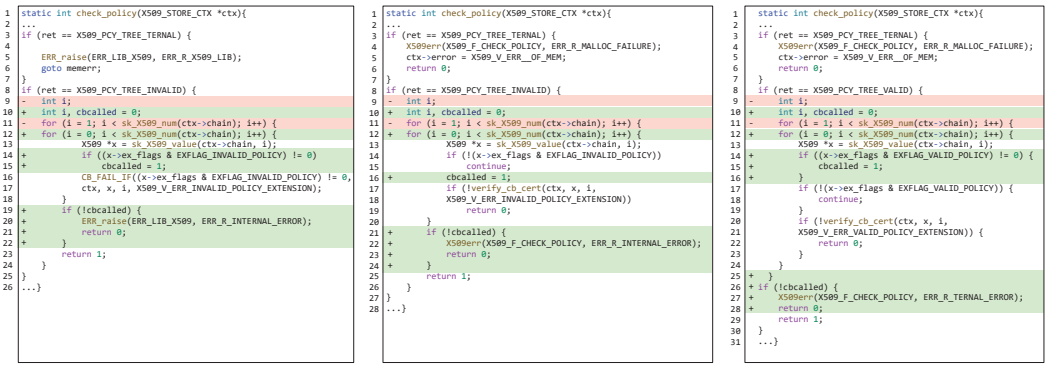
- We propose Mystique, a novel LLM-based approach for porting security patches. Mystique extracts semantic and syntactic signatures to drive LLM in generating a fixed function. It then checks both semantic and syntactic anomalies to guide LLM in refining the fixed function.
- Mystique outperforms the state-of-the-art approaches in patch porting success rate by at least 13.2% and 12.3% at both function and CVE levels, while also demonstrating strong generality across various scenarios, and successfully porting patches for 34 real-world vulnerable branches.

2 Motivation

Our investigation into vulnerability patch porting begins with a thorough analysis of state-of-the-art approaches. We used two CVEs to highlight their limitations and guide the design of Mystique.

2.1 Limitation of Pattern-Based Approaches

`CVE-2023-0465` [28] is a vulnerability in OpenSSL [32], caused by the improper handling of the `EXFLAG_INVALID_POLICY` flag within the `check_policy` function. It allows a security bypass due to incorrect validation in the certificate chain. Fig. 1(a) depicts the original patch in the `master` branch, where a sanity check (Lines 14-15) and a new error-handling block (Lines 19-22) were introduced. As shown in Fig. 1(b), developers manually adapted the patch, incorporating the sanity check (Line 16) and modifying the error-handling logic (Lines 21-24) to the `OpenSSL_1_1_1-stable` branch.



(a) Original patch of *check_policy* in Commit *e4142ec of Master* Branch

(b) Target patch of check_policy Ported by Human Developer in Commit *b013765* of *OpenSSL_1_1_1-stable* Branch

(c) Porting Patch of check_policy by *TSBPORT* of *OpenSSL_1_1_1-stable* Branch

Fig. 1. An Example of CVE-2023-0465, Porting Patch of master branch to function check_policy by Human Developers and TSBPORT in OpenSSL_1_1_1-stable Branch

Table 1. Success Rate of Patch Porting When PPatHF is Equipped with Our Key Modules

	PPatHF	PPatHF w/ Signature	PPatHF w/ Tuning	PPatHF w/ Refine	PPatHF w/ All
Succ. Rate (CVE)	0.385	0.516 (↑ 0.131)	0.641 (↑ 0.256)	0.531 (↑ 0.146)	0.840 (↑ 0.455)
Succ. Rate (Func.)	0.550	0.658 (↑ 0.108)	0.693 (↑ 0.143)	0.668 (↑ 0.118)	0.872 (↑ 0.322)

However, using `git cherry-pick` [12] to port the patch led to conflicts due to discrepancies between the branches, resulting in unresolved porting. Similarly, TSBPORT produced an erroneous patch, as shown in Fig. 1(c). It initially identified and applied the sanity check to Lines 14-16 in Fig. 1(c) before the redundantly existing sanity check (Lines 17-19). It incorrectly adapted the new error-handling logic into an invalid conditional block (Lines 25-28 in Fig. 1(c)), leading to a logic error that the function would never return 1, which disrupted the intended validation flow. FixMorph failed to handle the compilation process effectively despite we spent approximately 40 human hours adapting to OpenSSL. Other approaches like SkyPort and PatchWeave, which focus on injection-type vulnerabilities or test case-available vulnerabilities, also failed to port the patch due to the different vulnerability type and the absence of a test case for this vulnerability, respectively.

Those failures of pattern-based approaches highlight that the predefined patterns can restrict effectiveness and generality, limiting further patch adaption. In contrast, LLM-based approach PPatHF leverages the advanced code understanding capabilities of LLMs to port patches identical to human developers. Unlike pattern-based approaches, PPatHF can successfully port the patch while preserving the functional differences between `master` and `OpenSSL_1_1_1-stable` branch.

2.2 Limitation of LLM-Based Approach

Despite LLM's advanced capabilities, LLM-based approach PPatHF also faces critical limitations. We explored these limitations through `CVE-2023-1994` [29], a vulnerability in the `dissect_gquic_frame_type` function of Wireshark [55]. Since the patch is identical in both branches and the surrounding code is nearly the same, we avoid duplicating the porting results and only present the outcome in the target `release-3.6` branch in Fig. 2. If the pointer `gquic_info` is null, it can lead to a denial of service when passed to its callee function (Lines 24, 26 and 28 in Fig. 2). The human developers add a sanity check for `gquic_info` before its use in `master` branch of commit `ee314ac` and then in `release-3.6` branch of commit `8970fc1` (Lines 2-5 in Fig. 2).

The patches for the `master` branch and `release-3.6` branch are identical and simple, which was successfully ported by `git cherry-pick` and TSBPORT. However, PPatHF failed. The main reasons for this failure are three folds: ① Before porting, PPatHF identifies the entire compound block

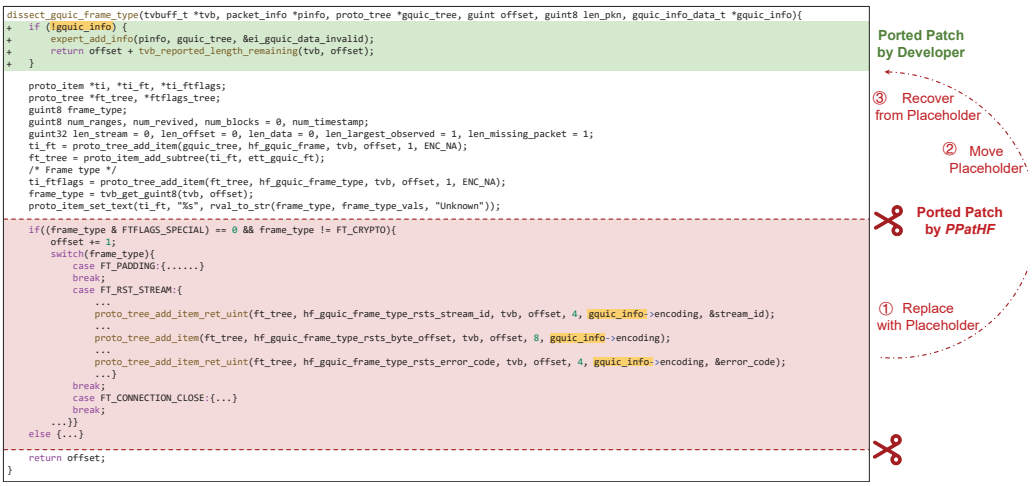


Fig. 2. An Example of CVE-2023-1994, Porting Patch to function dissect_gquic_frame_type by Human Developers (Green Hunk) and PPatHF (Red Hunk) in release-3.6 Branch

without modified lines in the original master branch and maps it to the target release-3.6 branch (Lines 17-34), replacing the block with a placeholder. However, this replacement also removes essential information, including critical function calls that attackers could exploit (e.g., Lines 24, 26, and 28), while leaving irrelevant statements intact (e.g., Lines 6-16), hindering accurate patch porting. ② During porting, PPatHF mistakenly moves the placeholder between Line 1 and Line 6 (Lines 2-5 are ported by human developers but are missing from the patch generated by PPatHF). This error arises because PPatHF relies heavily on the repository's history commits and their commit messages (i.e., intra-repository knowledge) for LLM fine-tuning, leading to misalignments with actual porting tasks and reduced effectiveness, particularly in unseen repositories. In this case, because LLM had not seen the placeholder during fine-tuning, the placeholder, which should remain in place, is incorrectly moved. ③ After porting, PPatHF recovers the previously moved code segment into the function, resulting in obvious syntactic errors (e.g., use before definition) and semantic errors (e.g., the dangerous use of the null pointer gquic_info remains unresolved after recovering). These issues leave the functionality broken and the vulnerability unfixed.

To address the limitation outlined in ①, we introduce a novel function slicing and completion technique that extracts signatures of semantically and syntactically relevant statements to accurately represent both the original patch function and the target vulnerable function. To mitigate ②, we construct the largest dataset of original-target patch function pairs across 33 popular repositories, designed to fine-tune an LLM with the same task of patch porting. To tackle the limitation described in ③, we design semantic and syntactic checks that guide LLM in refining patches more accurately.

We also enhance PPatHF by integrating Mystique's signature extraction module, fine-tuning strategy, and checking-refining module. As shown in Table 2, the original success rate of porting for 694 CVEs and 1,359 CVEs' functions are 0.385 and 0.550. Comparatively, Mystique can increase the performance by 34% to 66% at CVE level and by 20% to 26% at function level by integrating each module of Mystique. When integrating all the three critical modules to PPatHF, it has a significant increase of success rate at CVE and function level by 118% and 59%, respectively. The increment of PPatHF underscores the importance of how to drive LLM to generate high quality ported patches.

Table 2. Success Rate of Patch Porting When PPatHF is Equipped with Our Key Modules

	PPatHF	PPatHF w/ Signature	PPatHF w/ Tuning	PPatHF w/ Refine	PPatHF w/ All
Succ. Rate (CVE)	0.385	0.516 (↑ 0.131)	0.641 (↑ 0.256)	0.531 (↑ 0.146)	0.840 (↑ 0.455)
Succ. Rate (Func.)	0.550	0.658 (↑ 0.108)	0.693 (↑ 0.143)	0.668 (↑ 0.118)	0.872 (↑ 0.322)

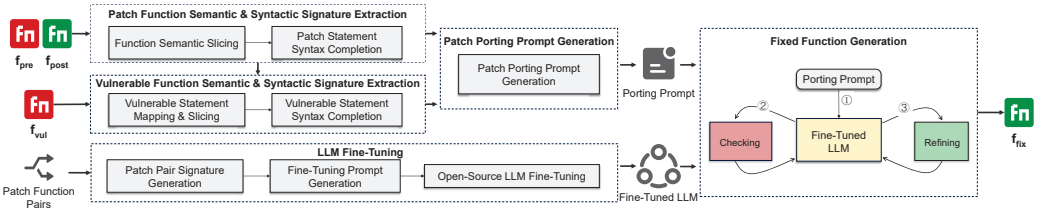


Fig. 3. Approach Overview of Mystique

3 Approach

The key idea of Mystique is to extract the semantic and syntactic vulnerable and fixed signatures of the patch from the original branch at function level, then extract the corresponding vulnerable function signature of the vulnerable function from the target branch at function level, guiding the fine-tuned large language model (LLM) to generate a fixed function. Since LLM may not always produce accurate results, Mystique incorporates an iterative checking and refining process, which enhances the success rate of patch porting. Finally, Mystique outputs the final fixed function. An overview of Mystique is shown in Fig. 3, which consists of five key modules.

- *Patch Function Semantic & Syntactic Signature Extraction.* Given a patch function from the original branch, Mystique extracts two vulnerability-relevant signatures, the pre-patch signature Sig_{pre} for pre-patch function f_{pre} and the post-patch signature Sig_{post} for post-patch function f_{post}.
- *Vulnerable Function Semantic & Syntactic Signature Extraction.* Given the vulnerable function f_{vul} from the target vulnerable branch, Mystique extracts the vulnerable function signature Sig_{vul}.
- *Patch Porting Prompt Generation.* Mystique applies the pre-patch signature Sig_{pre}, post-patch signature Sig_{post} and vulnerable function signature Sig_{vul} with a prompt template to generate the patch porting prompt, requiring LLM to satisfy essential constraints for the patch porting task.
- *LLM Fine-Tuning.* Mystique fine-tunes LLM by constructing original-target patch function pairs from two different branches. It extracts patch signatures (Sig_{opre}, Sig_{opost}) from the earlier original patched function and (Sig_{tpre}, Sig_{tpost}) from the later target patched function, using these signatures with the prompt template to generate the fine-tuning prompt to fine-tune LLM.
- *Fixed Function Generation.* Mystique generates an initial fixed signature Sig^0_{fix} using the generated prompt and fine-tuned LLM. This initial fixed signature is then iteratively refined after heuristic-based checking to address the potentially inaccurate patch generated by LLM. After this process, Mystique outputs the final fixed function f_{fix}.

3.1 Patch Function Semantic & Syntactic Signature Extraction

3.1.1 Function Semantic Slicing. Given the pre-patch function f_{pre} and the post-patch function f_{post}, Mystique first normalizes each statement in f_{pre} and f_{post} by removing all comments, tabs, and white spaces. This normalization step ensures that the patching process is tolerant of discrepancies in code formatting or comments. Next, Mystique uses the `git diff` command between f_{pre} and f_{post} to identify the set of deleted statements (denoted as $\mathbb{S}_{\text{del}}$) and the set of added statements (denoted as $\mathbb{S}_{\text{add}}$). Mystique then generates Code Property Graph (CPG) for both f_{pre} (denote as CPG_{pre}) and f_{post} (denote as CPG_{post}) using Joern [44]. The slicing process based on the generated CPGs is performed in two parts, i.e., *data-dependency slicing* and *control-flow slicing*.

Data-Dependency Slicing. MYSTIQUE iterates over each statement $s \in \mathbb{S}_{\text{del}}$ (resp. $s \in \mathbb{S}_{\text{add}}$) to perform data-dependency forward and backward slicing. This slicing traces the data dependencies of s forward and backward to the statements in CPG_{pre} (resp. CPG_{post}) that s depends on or is depended on. The slicing depth is set to sli_{dp}. This slicing captures how statements influence variable values and how those variables impact other statements. The resulting set of data-dependent statements is denoted as $\mathbb{S}_{\text{pre\_data}}$ (and $\mathbb{S}_{\text{post\_data}}$).

Control-Flow Slicing. MYSTIQUE performs control-flow slicing by iterating over each statement $s \in \mathbb{S}_{\text{del}}$ (resp. $s \in \mathbb{S}_{\text{add}}$) to extract the immediate dominate statements and the immediate post-dominates statement of s in CPG_{pre} (resp. CPG_{post}). This slicing helps to understand the execution flow of vulnerable and fixed statements. The resulting set is denoted as $\mathbb{S}_{\text{pre\_ctrl}}$ (resp. $\mathbb{S}_{\text{post\_ctrl}}$).

The sliced set of f_{pre} is $\mathbb{S}_{\text{pre\_ctrl}} \cup \mathbb{S}_{\text{pre\_data}}$, denoted as $\mathbb{S}_{\text{pre}}$. Similarly, the sliced set of f_{post} is $\mathbb{S}_{\text{post\_ctrl}} \cup \mathbb{S}_{\text{post\_data}}$, denoted as $\mathbb{S}_{\text{post}}$.

3.1.2 Patch Statement Syntax Completion. Maintaining correct syntax in these sliced code fragments is crucial for LLM's accurate understanding and processing while the data-dependency and control-flow slicing can lead to incomplete code fragments. Therefore, MYSTIQUE leverages Tree-sitter [49] to construct Abstract Syntax Tree (AST) that automatically recovers the syntactic integrity of the sliced code and provides a hierarchical representation of the code structure. Given a statement $s \in \mathbb{S}_{\text{pre}}$ (resp. $\in \mathbb{S}_{\text{post}}$), MYSTIQUE performs syntax completion and denotes these statements as $\mathbb{S}_{\text{pre\_ast}}$ (resp. $\mathbb{S}_{\text{post\_ast}}$). This process involves the following three steps.

Control Clause Completion. For a given statement $s \in \mathbb{S}_{\text{pre}}$ (resp. $\in \mathbb{S}_{\text{post}}$), MYSTIQUE traverses the AST's parent nodes of s, extracts the control statements (e.g., `if`, `else`, `switch`, `while`), and extracts the control sub-statements (e.g., the `case` statements corresponding to `switch`).

Exit Point and Jump Point Completion. For the extracted control statements and their sub-statements, MYSTIQUE extracts occurrences of `break`, `return` and `goto` in those control blocks, as these program exit points may contain important variable exit states.

Control Statement Bracket Completion. MYSTIQUE completes corresponding curly brackets for the extracted control statements and their sub-statements to maintain syntactic integrity.

MYSTIQUE treats all the extracted statements from AST as $\mathbb{S}_{\text{pre\_ast}}$ (resp. $\mathbb{S}_{\text{post\_ast}}$). Then the complete signature of f_{pre} (resp. f_{post}). is obtained, denoted as Sig_{pre} (resp. Sig_{post}), where, $Sig_{\text{pre}} = \mathbb{S}_{\text{pre}} \cup \mathbb{S}_{\text{pre\_ast}}$ (resp. $Sig_{\text{post}} = \mathbb{S}_{\text{post}} \cup \mathbb{S}_{\text{post\_ast}}$). All the other statements in f_{pre} (resp. f_{post}) are considered as removable statements. Following PPATHF [34], MYSTIQUE replaces them with placeholders, denoted as $\mathbb{P}_{\text{pre}}$ (resp. $\mathbb{P}_{\text{post}}$).

3.2 Vulnerable Function Semantic & Syntactic Signature Extraction

3.2.1 Vulnerable Statement Mapping & Slicing. MYSTIQUE first generates a Code Property Graph (CPG) for f_{vul} (denote as CPG_{vul}) using Joern [44] and then maps the deleted statements $\mathbb{S}_{\text{del}}$ in f_{pre} to their corresponding vulnerable statements in f_{vul}, resulting in a set of mapped statements, which is then used for slicing. The mapping process involves the following three steps.

(1) Given a deleted statement $s \in \mathbb{S}_{\text{del}}$, MYSTIQUE first calculates the syntactic similarity with each statement in f_{vul} using the Levenshtein distance [54]. If the similarity exceeds a predefined threshold th_{ld}, a mapping between s and the statement in f_{vul} is established. The default threshold is set to 0.55, following previous work [7]. Since there may be multiple candidate mapped statements, MYSTIQUE denotes the candidate statements in f_{vul} as $\mathbb{S}_{\text{cand}}$.
(2) MYSTIQUE then performs forward and backward data-dependency and control-dependency slicing for s and its mapped statement in $\mathbb{S}_{\text{cand}}$ to extract the statement set that has data or

control dependencies with them in CPG_{pre} and CPG_{vul}. It calculates similarity between the data-dependency and control-dependency statements of s and each candidate statement in $\mathbb{S}_{\text{cand}}$. The candidate statement with the highest similarity score is selected as the mapped statement.

(3) If no statement meets the mapping criteria in (1), MYSTIQUE iteratively repeats Steps 1 and 2 for the dominate and post-dominate statements of s until it reaches the entry and exit points of the function. The statements between mapped dominate and post-dominate are selected as the mapped statement hunk.

The resulting mapped statements, along with the statement hunks between the mapped dominate and post-dominate, are denoted as $\mathbb{S}_{\text{map}}$. If the patch has no deleted statements, MYSTIQUE regards $Sig_{\text{post}} \setminus \mathbb{S}_{\text{add}}$ as the vulnerable statement, and then conducts the same mapping process. Next, similar to Section 3.1, MYSTIQUE performs function slicing (see Sec. 3.1.1) on $\mathbb{S}_{\text{map}}$. The final sliced set of f_{vul} is $\mathbb{S}_{\text{vul\_data}} \cup \mathbb{S}_{\text{vul\_ctrl}}$, denoted as $\mathbb{S}_{\text{vul}}$.

3.2.2 Vulnerable Statement Syntax Completion. The syntax completion for vulnerable statements follows the same process as described in Sec. 3.1.2 for patch statement syntax completion. The final signature of f_{vul}, denoted as Sig_{vul}, is obtained, i.e., $Sig_{\text{vul}} = \mathbb{S}_{\text{vul}} \cup \mathbb{S}_{\text{vul\_ast}}$.

3.3 Patch Porting Prompt Generation

Fig. 4 shows the prompt template for patch porting and fine-tuning. It consists of three components, i.e., task description, constraint instruction, and input. The patch porting prompt is denoted as P_{port}.

- **Task Description** defines the description of the porting task. `%Language%` specifies the programming language (e.g., C, Java) of the task. The description informs LLM that it must adapt a patch to fix a vulnerable function while adhering to the language-specific characteristics.
- **Constraint Instruction** outlines a set of constraints that guide LLM during the patch porting process. Specifically, **C1** instructs LLM to focus on adapting the patch as necessary, acknowledging that the patch may not directly apply to the vulnerable function. **C2** ensures that LLM only makes changes necessary to apply the patch without introducing additional fixes or improvements, and it maintains the original functionality to the greatest extent possible. **C3** instructs LLM to retain the special annotation (placeholders) untouched, preserving their integrity. **C4** advises LLM not to fill in any missing parts of the code, as this could introduce potential syntactic errors. LLM should output only the fixed code, even if incomplete portions are present.
- **Input** defines the format for the provided patch and vulnerable function. The patch signature (Sig_{pre}, Sig_{post}) and the vulnerable function signature Sig_{vul} are given as inputs. To reduce token consumption and highlight discrepancies, MYSTIQUE converts the patch signature to a diff format, denoted as `%diff(`Sig_{pre}, Sig_{post}`)%`. MYSTIQUE fills `%`Sig_{vul}`%` with the vulnerable signature Sig_{vul} to indicate the target vulnerable function that requires patch porting.

3.4 LLM Fine-Tuning

To enhance LLM's capability in performing patch porting, we employ a fine-tuning process specifically tailored to this task. Unlike PPATHF, where the fine-tuning task is not aligned with the patch poring task, MYSTIQUE ensures that the fine-tuning task is fully aligned with the patch porting task, leading to more accurate and effective results.

3.4.1 Patch Pair Signature Generation. We first construct the dataset of original-target patch function pairs (see Sec. 4.1) and split each pair into the original patch function (denoted as f_{opre} and f_{opost}) and the target patch function (denoted as f_{tpre} and f_{tpost}). Given f_{opre}, f_{opost} and f_{tpre} (serving a similar role as f_{pre}, f_{post} and f_{vul} in patch porting task), we generate the signatures denoted as Sig_{opre}, Sig_{opost} and Sig_{tpre} (serving a similar role as Sig_{pre}, Sig_{post} and Sig_{vul} in patch porting

Task Description

You are a professional and cautious **%Language%** programmer.
I will give you a patch and a vulnerable function.
Your task is to refer to the patch to fix the vulnerable function, and output the fixed function.

Constraint Instruction

Here are the constraints for your output:
C1. Need to make adaptation for the patch I gave you. It may not be directly applicable to the vulnerable function.
C2. Do not make any other fixes or improvements, you only need to adapt and fix patch parts.
C3. Do not delete or add any placeholders in the code.
C4. Do not fill in the missing parts, which will cause potential syntactic error issues. You may notice that there are some missing parts in the code I gave you, but it's okay. You just need to output the fixed code.

Input

Original Function Patch: **%diff(Sig_{pre}, Sig_{post})%**
Vulnerable Function: **%Sig_{vul}%**

Fig. 4. Overview of Prompt Template

Table 3. Mapping of Constraints, Anomalies and Prompts

Constraints	Anomalies	Prompts
C1	**A1** Semantic Fixing Anomalies	**P_a** Your patch porting result don't meet **%*Cn*%**, please refine your fixed function.
C2		
C3	**A2** Placeholder Anomalies	
C4	**A3** Syntactic Fixing Anomalies	**P_b** Your patch porting result obeys **% *syntactic error* %**, please refine your fixed function.

task) for fine-tuning LLM. MYSTIQUE fine-tunes LLM in a supervised way. Therefore, following Sec. 3.2, MYSTIQUE maps the added statements in f_{opost} to their corresponding fixed statements in f_{tpost}, resulting in a set of mapped statements, and then further generates Sig_{tpost} after slicing and completion, which serves as the expected output for the supervised fine-tuning task.

3.4.2 Fine-Tuning Prompt Generation. MYSTIQUE generates the fine-tuning prompt with the same prompt template of the porting task using Sig_{opre}, Sig_{opost} and Sig_{tpre}, denoted as P_{ft}.

3.4.3 Open-Source LLM Fine-Tuning. For each function pair, MYSTIQUE defines Sig_{tpost} as the expected $Output$, and constructs $(P_{\text{ft}}, Output)$ for LoRA fine-tuning [14]. We select a pre-trained model, and apply low-rank decomposition to model weights $W \approx A \times B$. This enables efficient adaptation using smaller matrices $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times d}$. During fine-tuning, the model generates Sig'_{tpost}, and the loss $\mathcal{L}(Sig_{\text{tpost}}, Sig'_{\text{tpost}})$ is computed and minimized using the AdamW optimizer [22]. This process iterates until the loss stabilizes, yielding the fine-tuned LLM $\mathcal{M}$.

3.5 Fixed Function Generation

The process of generating the fixed function relies on the prompt P_{port} from Sec. 3.3 and the fine-tuned LLM $\mathcal{M}$ from Sec. 3.4. MYSTIQUE first generates an initial fixed signature and then iteratively checks and refines it. This iterative process continues until the desired conditions are met and then MYSTIQUE outputs the final fixed function f_{fix}.

3.5.1 Initial Signature Generation. Given a porting prompt P_{port} and a fine-tuned LLM $\mathcal{M}$, MYSTIQUE generates an initial output Sig^0_{fix}, where $Sig^0_{\text{fix}} = \mathcal{M}\left(P_{\text{port}}\right)$.

3.5.2 Heuristic-Based Checking. Based on the constraints of P_{port} (i.e., C1-4), MYSTIQUE identifies three types of anomalies that prevent LLM from correctly generating the fixed function, as shown in Table 3. The constraints C1-4 are designed to guide LLM in porting patch successfully, without breaking the functionality or introducing anomalies. We design three types of anomalies that violate the constraints, which leads to the semantic error or syntactic error of the fixed function. The three main anomalies are designed and detected as follows.

Semantic Anomalies (A1). C1 of the prompt template specifies that the ported patch should make necessary adjustments, while C2 emphasizes that the patch must preserve the functionality of the target branch. Failing to strike the right balance can result in semantical failure after patch porting: under-fixing violates C1 by not applying sufficient changes, while over-fixing violates C2 by altering the intended functionality. MYSTIQUE first calculates the edit distance [54] between Sig_{pre} and Sig_{vul}, denoted as $Sim(Sig_{\text{pre}}, Sig_{\text{vul}})$. It then calculates the edit distance between Sig_{post} and Sig_{fix}, denoted as $Sim(Sig_{\text{post}}, Sig_{\text{fix}})$. If $Sim(Sig_{\text{pre}}, Sig_{\text{vul}})$ is significantly greater than $Sim(Sig_{\text{post}}, Sig_{\text{fix}})$, we consider the patch porting as over-fixed. For example, if the pre-patch function and the vulnerable function are the same, but ported patch is significantly different from the post-patch function, this indicates that LLM has over-fixed, potentially breaking the functionality of the target branch. Conversely, if $Sim(Sig_{\text{post}}, Sig_{\text{fix}})$ is significantly greater than $Sim(Sig_{\text{pre}}, Sig_{\text{vul}})$, we consider the patch porting as under-fixed. For example, if the pre-patch function and the vulnerable function are significantly different, but the ported patch remains unchanged with the post-patch function, this indicates that LLM has under-fixed and failed to adapt the patch properly. We set thresholds for over-fixing as th_{ov} and under-fixing as th_{ud}. If $Sim(Sig_{\text{post}}, Sig_{\text{fix}})$ - $Sim(Sig_{\text{pre}}, Sig_{\text{vul}}) > th_{ov}$ or $Sim(Sig_{\text{pre}}, Sig_{\text{vul}})$ - $Sim(Sig_{\text{post}}, Sig_{\text{fix}}) > th_{ud}$, anomaly A1 is detected.

Placeholder Anomalies (A2). After slicing, there are special *placeholders* that LLM should not add, delete or change place. However, we observe that LLM often struggles to handle placeholders, leading to anomalies, which violates C3. Given Sig_{vul}, MYSTIQUE anchors each placeholder with its left sibling (parent node) and right sibling (child node) in its AST. If both siblings exist in Sig^0_{fix} but the placeholder is added, deleted or changed place, anomaly A2 is detected.

Syntactic Anomalies (A3). Certain anomalies cannot be detected by relying solely on code signatures. They require a comprehensive understanding of the entire fixed function, as mentioned in Sec. 2.2. These anomalies, introduced by function signatures, can disrupt C4 and lead to syntactic failures after patch porting. To address this, MYSTIQUE first recovers the code hunks corresponding to each placeholder. After restoring all placeholders along with their related statements, MYSTIQUE first generates the initial fixed function f^0_{fix}. Next, MYSTIQUE uses Clang-tidy [5], a powerful tool for identifying syntactic anomalies in f^0_{fix}, to detect syntactic errors. To minimize false positives, MYSTIQUE ignores the syntactic errors identified in both the vulnerable function f_{vul} and the initial fixed function f^0_{fix} and if Clang-tidy detects any other syntactic errors, anomaly A3 is flagged.

3.5.3 Refining. MYSTIQUE uses $\mathcal{M}$ to refine its most recent output. As shown in Table 3, if A1 or A2 is detected, MYSTIQUE uses the prompt P_a to guide LLM in generating a refined result. Here, %Cn% corresponds to the specific constraint being addressed, C1 for under-fixing in A1, C2 for over-fixing in A1, and C3 for A2. If anomaly A3 is detected, MYSTIQUE uses the prompt P_b to guide LLM in generating the refined result. Here, %syntactic error% represents the potential syntactic errors identified by Clang-tidy.

3.5.4 Iterating Checking and Refining. MYSTIQUE iteratively applies the process of checking and refining. To maintain context, the history of previous checks and refinements is appended to the prompt. Initially, it detects A1 and A2 anomalies before recovering placeholders, and conducts the corresponding refining as shown in Eq. 1. If no A1 or A2 anomalies are found or the prompt token

limit tk_{num} is achieved, MYSTIQUE outputs the fixed signature Sig_{fix}. MYSTIQUE then proceeds to recover the placeholders along with their code hunks in Sig_{fix} to generate intermediate f_{fix}. As shown in Eq. 2, if the prompt does not exceed the token limit and anomaly A3 is detected, MYSTIQUE includes A3 in the prompt for refining until no A3 is found or the prompt token limit tk_{num} is achieved. The final output f_{fix} is the completed fixed function for the target branch.

$$Sig_{\text{fix}}^{t+1} = \mathcal{M}\left(Sig_{\text{fix}}^{t}, P_a\right) \tag{1}$$

$$f_{\text{fix}}^{t+1} = \mathcal{M}\left(f_{\text{fix}}^{t}, P_b\right) \tag{2}$$

4 Evaluation

We implement MYSTIQUE with 7K lines of Python code and design the following research questions.

- **RQ1: Effectiveness Evaluation.** How effective is MYSTIQUE in porting patches?
- **RQ2: Ablation Study.** How does each component contribute to MYSTIQUE?
- **RQ3: Parameter Sensitivity.** How do the parameters affect the effectiveness of MYSTIQUE?
- **RQ4: Generality Evaluation.** How is the generality of MYSTIQUE in different scenarios?
- **RQ5: Efficiency Evaluation.** How is the efficiency of MYSTIQUE?
- **RQ6: Usefulness Evaluation.** How is the practical usefulness of MYSTIQUE?

4.1 Evaluation Setup

Ground Truth. Following TSBPORT [62], we choose C projects to construct the ground truth. The ground truth is established through a rigorous four-step examination process.

(1) *Collecting Vulnerabilities and Patches.* We collected vulnerabilities from CVE/NVD [27], along with their corresponding repository patch references (i.e., GitHub commits) between January 2014 and May 2024. Then, we filtered those vulnerabilities whose corresponding repositories are not written in C. As a result, we gathered 7,514 CVEs with their patches.

(2) *Expanding Vulnerabilities and Patches.* To expand the vulnerabilities, we incorporated the vulnerability dataset used by TSBPORT [62]. To expand patches of each vulnerability, we leveraged the collected patches to identify additional potential patches. Specifically, for each collected commit, we scanned all commits in the corresponding repository, and expanded the patch set if they met one of the following four criteria, **(a)** The patch code is identical to one of the collected commits. **(b)** The edit distance similarity score exceeds 0.5 with at least one of the collected commits, and the commit messages are identical. **(c)** The commit message references the SHA value of a collected commit. **(d)** The commit message references a collected CVE ID. Following this augmentation, we collected a total of 7,532 CVEs with their 15,478 patches.

(3) *Selecting CVE Patch Pairs.* Following PPATHF [35], we manually filtered commits unrelated to the vulnerability, and further refined the commits that involved files unrelated to C or modifications outside functions. After filtering, we sorted all commits by timestamp, and selected the earliest and latest commit for each vulnerability to form a CVE patch pair. The earliest commits are regarded as the original patch and the latest commits are regarded as the target patch. If the vulnerability only contains one single patch, we also excluded it. After this process, we finalized CVEs with their corresponding patch pairs.

(4) *Splitting CVE Patch Pairs.* We used patch pairs dated after the release of CodeLLama on August 23, 2023 for patch porting task, and patch pairs dated before this release date for fine-tuning task, ensuring no data leakage between fine-tuning set and patch porting set. We then split each patch into function quadruples (f_{pre}, f_{post}, f_{vul}, $f_{\text{human-fix}}$) (resp., (f_{opre}, f_{opost}, f_{tpre}, f_{tpost})) for each function for the patch porting task (resp., then fine-tuning task).

Two of the authors were involved in filtering commits unrelated to the vulnerability when *Selecting CVE Patch Pairs*. Each author had over 5 years of experience in software security. We measured their agreement using Cohen's Kappa coefficient, which reached 0.981 for inspecting vulnerability-unrelated commits. A third author was involved in resolving disagreements. As a result, we collected 3,536 CVEs with their 5,977 patch pairs at function level across 33 different repositories, 694 CVEs with their 1,359 patch pairs at function level for the patch porting task and 2,842 CVEs with their 4,618 patch pairs at function level for the fine-tuning task.

Fine-Tuned LLM Selection. We use CodeLlama [26], a recent and powerful foundation large model designed for general code synthesis and understanding, to implement the porting module of Mystique. We download the public accessible pretrained weights (with 13B parameters and instruct-tuned) from the Hugging Face Transformer library to initialize our model.

During the fine-tuning stage, we adopt LoRA (Low-Rank Adaptation) [14] to tune LLM. We set `lora_r` to 16 and `lora_alpha` to 32 for reducing memory requirement according to Raschka's work [37], with less time consumption and better performance. In addition, we set `batch size` to 8, `epochs` to 10 and `learning rate` to 3e-4, following PPatHF [35].

During the patch porting stage, we configure the maximum context tokens to 4,096. This extended token limit accommodates the longer function and more round of refining, ensuring that input and output will not be truncated. Besides, `temperature` and `top_p` affect the consistency of outputs of the fine-tuned LLM. We choose to change temperature instead of changing both of them, following the practice in prior work [60]. Here, we set temperature to 0.5, ensuring that the model maintains consistency in patch porting while giving creative inferences.

Evaluation Metrics. The metrics of Exactly Match (E-M) and Success (Succ) are used in TSBPORT [62] to evaluate patch porting quality. E-M refers to the automatically ported patches that are identical to the manually ported patch, while Succ refers to automatically ported patches that are semantically equivalent to the manually ported patches. Both metrics are shown in the form of x (y), where x is the number of patches that are E-M or Succ, and y is the E-M rate or Succ rate. We apply both E-M and Succ to evaluate patch porting performance at the function and CVE level, providing a comprehensive assessment of porting quality. Additionally, patches that do not meet the semantically equivalence are categorized as failures (Fail), denoted in the same form of x (y), where x represents the number of failed patches and y indicates the failure rate. In addition, we introduce Code BLEU (C-B) [39] as an additional metric to quantify the difference between failed patches and the human ported patches with similarity values ranging from 0 to 1, where a higher value indicates greater similarity. We choose Code BLEU because, unlike binary metrics like E-M or Succ, it captures the degree of similarity in partially correct patches, offering a more nuanced view of how close failed patches are to the correct ones.

Baselines. We include seven state-of-the-art approaches, including pattern-based approaches (FixMorph [41] and TSBPORT [62]), LLM-based approach (PPatHF [35]), the most recently closed-source LLMs, i.e., GPT-3.5 [30] and GPT-4o [31], and open-source LLMs, i.e., CodeLlama with 13 billion parameters and instruct-tuned, and StarCoder with 15.5 billion parameters used in PPatHF [35]. According to PPatHF [34], FixMorph only successfully ports about 4% of patches in the repositories other than Linux, highlighting its limited performance. Additionally, compiling data from diverse C repositories requires substantial effort. Given these constraints, we restrict our evaluation of FixMorph within fine-tuning sub-dataset from Linux repositories, referred to as FixMorph (L.). Similarly, we extract results of Mystique from Linux, denoted as Mystique (L.).

4.2 Effectiveness Evaluation (RQ1)

RQ1 Setup. We assessed Mystique using the ground truth, and compared its performance with baseline approaches. We first fine-tuned Mystique with the fine-tuning dataset of our ground

Table 4. Results of Our Effectiveness Evaluation Compared to the State-of-the-Art

Tool	# CVE	# Func.	Success (CVE Level)		Success (Func. Level)		Failure (Func. Level)	
			E-M	Succ	E-M	Succ	Fail	C-B
FixMorph (L.)	513	1,037	97 (0.189)	145 (0.283)	96 (0.093)	151 (0.146)	886 (0.652)	0.858
Mystique (L.)			**479 (0.934)**	**484 (0.943)**	**999 (0.963)**	**1,004 (0.968)**	**33 (0.032)**	**0.921**
TSBPORT	694	1,359	557 (0.803)	571 (0.823)	1,127 (0.829)	1,145 (0.843)	214 (0.157)	0.919
PPatHF			253 (0.365)	267 (0.385)	727 (0.535)	747 (0.550)	612 (0.450)	0.876
GPT-3.5			205 (0.295)	219 (0.316)	560 (0.412)	595 (0.438)	764 (0.562)	0.783
GPT-4o			488 (0.703)	497 (0.716)	1,082 (0.796)	1,094 (0.805)	265 (0.195)	0.859
StarCoder			229 (0.330)	242 (0.349)	656 (0.483)	676 (0.497)	683 (0.503)	0.874
CodeLlama			338 (0.487)	342 (0.492)	858 (0.631)	869 (0.639)	490 (0.361)	0.878
Mystique			**628 (0.905)**	**641 (0.924)**	**1,283 (0.944)**	**1,297 (0.954)**	**62 (0.046)**	**0.921**

truth. PPatHF, however, was fine-tuned with the given commit message and its patch. Given the millions of commits across the 33 collected popular repositories, we sampled 5% of commits from the Linux repositories and 10% from the other 32 repositories. We then formatted the project history commits to include single-function changes between January 2014 and May 2024, aligning with our fine-tuning dataset selection. We further filtered the commits to include only those before the release time of StarCoder (i.e., July 1, 2022) to prevent data leakage. As a result, the fine-tuning dataset for PPatHF was composed of 23,561 commits from Linux and 7,913 commits from the other repositories, which was four times larger than the original dataset used for PPatHF.

Overall Result. Table 4 shows the overall effectiveness results on the patch porting dataset. Mystique has shown superior performance across all metrics, leading in both CVE and function level. Specifically, it achieves 628 (0.905) of E-M and 641 (0.924) of Succ at CVE level and 1,283 (0.944) of E-M and 1,297 (0.954) of Succ at function level, surpassing all baselines approaches. C-B, which assesses the closeness of failed patches to the true patch, is highest for Mystique at 0.921, indicating that even the failed cases are more similar to the true patches compared to other approaches. Comparatively, the best state-of-the-art tool, TSBPORT has 557 (0.803) of E-M and 571 (0.823) of Succ at CVE level and 1,127 (0.829) of E-M and 1,145 (0.843) of Succ at function level, with a C-B of 0.919. Mystique not only improves in the Succ rate by 0.101 (12.3%) and 0.111 (13.2%) at CVE and function level, but also slightly outperforms in C-B by 0.002. The LLM-based approach PPatHF shows weaker performance with a Succ rate of 0.385 at CVE level and 0.550 at function level, and a C-B of 0.876. Mystique markedly outperforms it in the Succ rate at CVE and function level by 0.539 (140%) and 0.404 (73.5%), and also outperforms it in C-B by 0.045, highlighting the effectiveness of Mystique. The best LLM GPT-4o shows a Succ rate of 0.716 at CVE level and 0.805 at function level, with a C-B of 0.859. Mystique significantly surpasses GPT-4o in the Succ rate by 0.208 (29.1%) and 0.149 (18.5%) at CVE and function level, respectively, but also outperformed it in C-B by 0.062, indicating the limitation of the porting patches by LLM without any process.

In-Depth Analysis. As shown in Table 5, there are 62 functions that Mystique fails to patch. The majority of failures are signature-related (35 failures), primarily due to inter-procedural insensitivity, macro handling limitations, and token overflow. Moreover, tool-related issues account for 6 failures, all stemming from Joern slicing inaccuracies. LLM-related issues contribute to 19 failures, including placeholder mishandling, hallucination, and type insensitivity. Besides, branch-related issues lead to 2 failures, both caused by significant discrepancies.

Contextual loss during slicing is a major root cause in these failures. The most significant source is inter-procedural insensitivity (18 failures), where Mystique fails to capture security-critical relationships between callers and callees. Macro handling limitations further contribute to contextual loss (14 failures). Tool-specific constraints in Joern and Tree-sitter, particularly around complex pointer operations and alias analysis, account for an additional 6 failures. While static intra-procedural data/control dependency slicing and AST completion provide a solid foundation, they occasionally

Table 5. In-Depth Analysis of Key Issues by Category

Category	Key Issues	Failures	Category	Key Issues	Failures
Signature-Related	Inter-procedural insensitivity	18	LLM-Related	Placeholder mishandling	9
	Macro handling limitations	14		Hallucination	8
	Token overflow	3		Type insensitivity	2
Tool-Related	Joern slicing inaccuracies	6	Branch-Related	Significant discrepancies	2

Table 6. Results of Our Ablation Study

Ablation	Success (CVE Level)		Success (Func. Level)		Failure (Func. Level)	
	E-M (Δ E-M)	Succ (Δ Succ)	E-M (Δ E-M)	Succ (Δ Succ)	Fail (Δ Fail)	C-B (Δ C-B)
w/o Signature	0.717 (↓ 0.188)	0.729 (↓ 0.195)	0.796 (↓ 0.148)	0.802 (↓ 0.152)	0.198 (↑ 0.152)	0.803 (↓ 0.118)
w/o Tuning	0.438 (↓ 0.467)	0.444 (↓ 0.480)	0.567 (↓ 0.377)	0.575 (↓ 0.379)	0.425 (↑ 0.379)	0.883 (↓ 0.038)
w/o Refine	0.813 (↓ 0.092)	0.817 (↓ 0.107)	0.889 (↓ 0.055)	0.892 (↓ 0.062)	0.108 (↑ 0.062)	0.901 (↓ 0.020)

fail to preserve full semantic context. Furthermore, type insensitivity (2 failures) introduces risks in security patches, where type changes may inadvertently introduce vulnerabilities. The remaining failures arise from LLM hallucinations (8 failures), placeholder mishandling (9 failures), and branch discrepancies (2 failures). Addressing these problems requires advancements in LLM-based code understanding and improvements in tool precision. Despite these limitations, Mystique achieves a 95.4% success rate at the function level, demonstrating its strong foundation. The identified areas for improvement provide a clear path for enhancing Mystique's effectiveness further.

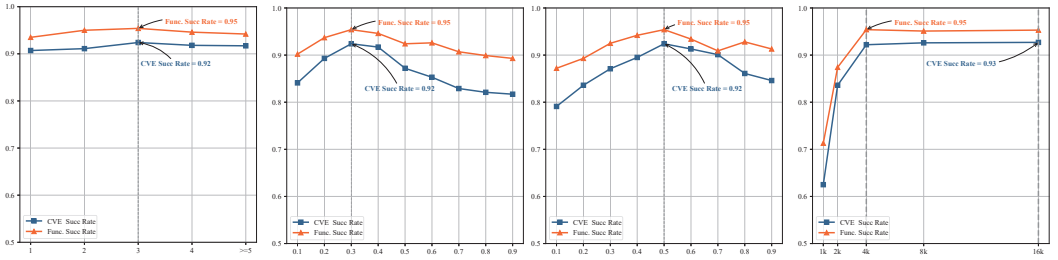
Summary: Mystique demonstrates outstanding effectiveness in patch porting, achieving 641 (0.924) of Succ at CVE level and 1,297 (0.954) of Succ at function level, which outperforms the best state-of-the-art TSBPORT at the Succ rate by 0.101 (12.3%) and 0.111 (13.2%) at CVE and function level, respectively. Mystique also achieves the highest C-B of 0.921, indicating greater similarity to true patches even in failed cases, which underscores its superior capability in accurately addressing and refining patches compared to state-of-the-art approaches.

4.3 Ablation Study (RQ2)

RQ2 Setup. We created three ablated versions of Mystique, i.e., (a) keeping the pre-patch function, post-patch function and vulnerable function without slicing and completion, thereby ablating the patch signature and vulnerable signature (w/o Signature); (b) ablating the fine-tuning from Mystique (w/o Tuning); and (c) ablating checking and refining (w/o Refine).

Overall Result. Table 6 presents the results of our ablation study. Overall, E-M and Succ decrease across the three ablated versions. Specifically, it exhibits the most significant drop in Mystique w/o Tuning, with E-M and Succ rate of 0.467 and 0.480 at CVE level, with E-M and Succ rate of 0.377 and 0.379 at function level, and a C-B drop of 0.038. This decline is primarily because the untuned LLM interprets sliced statements as syntactically incomplete fragments rather than meaningful semantic units, causing conflicts with Mystique's placeholder preservation mechanism and leading to failures. Additionally, without fine-tuning, LLM treats placeholders as standard code annotations rather than structural markers, further reducing its effectiveness. As a result, its performance falls even below that of the untrained CodeLlama in Table 4, highlighting the necessity of fine-tuning.

Besides, Mystique w/o Signature suffers the second-largest E-M and Succ rate drop of 0.188 and 0.195 at CVE level, and 0.148 and 0.152 at function level, and a C-B drop of 0.118. Additionally, Mystique w/o Refine results in decreased E-M rate and Succ rate by 0.092 and 0.107 at CVE level and 0.055 and 0.062 at function level. The result indicates that all the three ablated components of Mystique make important contribution to its effectiveness.



(a) sli_{dp} (b) th_{ov} (c) th_{ud} (d) tk_{num}

Fig. 5. Results of Our Sensitivity Analysis

Summary: Removing any component of MYSTIQUE results in noticeable effectiveness drops. Specifically, ablating each component can suffer the Succ rate drop from 0.107 to 0.480 at CVE level, and from 0.062 to 0.379 at function level.

4.4 Parameter Sensitivity (RQ3)

RQ3 Setup. Four key parameters are configurable in MYSTIQUE, namely the slicing depth (sli_{dp}), the over-fixing threshold (th_{ov}), the under-fixing threshold (th_{ud}), and the maximum session token limit (tk_{num}). The default values for these parameters are set to 3, 0.3, 0.5, and 4,096 (4K), respectively. To assess the sensitivity of these parameters on MYSTIQUE's success rate, we systematically varied one parameter while keeping the other three unchanged.

Overall Result. Fig. 5 presents the results of our sensitivity analysis. First, a deeper slicing depth can cause the generated fixed function to include more irrelevant statements, whereas a shallower slicing depth might miss critical statements. MYSTIQUE achieves the best balance with a slicing depth of 3. This threshold is crucial for detecting the appropriate degree of patch adaptation. Second, the best performance is observed when th_{ov} and th_{ud} are set to 0.3 and 0.5, respectively. Last, MYSTIQUE demonstrates a significant performance improvement at 4K tokens, with only a slight increase at 16K tokens in Succ rate at CVE level. Consequently, the maximum token length is set to 4K to reduce memory usage and computational costs associated with longer token lengths.

Summary: MYSTIQUE achieves the optimal performance when sli_{dp}, th_{ov}, th_{ud} and tk_{num} are set to 3, 0.3, 0.5 and 4,096, respectively.

4.5 Generality Evaluation (RQ4)

RQ4 Setup. To evaluate the generality of MYSTIQUE, we designed four generality experiments by applying MYSTIQUE across different LLMs, projects, bugs, and programming languages.

Cross-LLM Porting Evaluation. To assess the impact of different underlying large language models (LLMs) on patch porting, we conducted a cross-LLM porting evaluation by aligning the LLMs used in PPATHF and MYSTIQUE. We replaced CODELLAMA with STARCODER in MYSTIQUE, and replaced STARCODER with CODELLAMA in PPATHF. All models were fine-tuned with the same hyperparameters (e.g., learning rate, and LoRA settings) across both approaches to ensure a fair comparison.

Cross-Project Porting Evaluation. We divided our datasets into two categories, Linux (L.) and Others (O.). MYSTIQUE was fine-tuned separately on L. and O., with patch porting conducted on the opposite category (i.e., fine-tuning on L., porting on O. (L./O. group) and vice versa (O./L. group)). This allows us to assess how MYSTIQUE generalizes across different projects by comparing against PPATHF.

Bug Porting Evaluation. To evaluate MYSTIQUE's effectiveness on general C bugs, we curated a patch porting dataset consisting of common C bugs. The dataset was constructed through a meticulous process. (1) We chose the top 100 C repositories on GitHub based on star count, ensuring active

Table 7. Results of Our Generalization Evaluation Across Different LLMs

Tool	# CVE	# Func.	Success (CVE Level)		Success (Func. Level)		Failure (Func. Level)	
			E-M (Δ E-M)	Succ (Δ Succ)	E-M (Δ E-M)	Succ (Δ Succ)	Fail	C-B
PPatHF-StarCoder	694	1359	253 (0.365)	267 (0.385)	727 (0.535)	747 (0.550)	612 (0.450)	0.876
Mystique-StarCoder			478 (0.689)	493 (0.710)	1080 (0.795)	1095 (0.806)	264 (0.194)	0.896
PPatHF-CodeLlama			345 (0.497)	361 (0.520)	888 (0.653)	910 (0.670)	449 (0.330)	0.846
Mystique-CodeLlama			**628 (0.905)**	**641 (0.924)**	**1283 (0.944)**	**1297 (0.954)**	**62 (0.046)**	**0.921**

Table 8. Results of Our Generalization Evaluation Across Different Projects

Group	Tool	# CVE (Func.)	Success (CVE Level)		Success (Func. Level)		Failure (Func. Level)	
			E-M (Δ E-M)	Succ (Δ Succ)	E-M (Δ E-M)	Succ (Δ Succ)	Fail	C-B
L./O.	PPatHF	181 (322)	42 (0.232)	44 (0.243)	110 (0.342)	115 (0.357)	207 (0.643)	0.749
	Mystique		**150 (0.829)**	**152 (0.840)**	**286 (0.888)**	**288 (0.894)**	**34 (0.106)**	**0.878**
O./L.	PPatHF	513 (1,037)	129 (0.251)	131 (0.255)	459 (0.443)	464 (0.447)	573 (0.553)	0.782
	Mystique		**469 (0.914)**	**470 (0.916)**	**984 (0.949)**	**985 (0.950)**	**52 (0.050)**	**0.915**

maintenance by requiring at least two branches with activities within August 2024. (2) Commits containing keywords like "update" or "CVE" were excluded to avoid bug-unrelated updates. We further reduced the dataset by randomly selecting 5% of the commits made between January 1, 2024, and August 1, 2024. (3) The filtered commits were manually reviewed, resulting in 652 confirmed bug-fixing commit pairs for the final bug porting dataset.

Java Vulnerability Porting Evaluation. To extend the evaluation to another programming language, we constructed a Java vulnerability patch porting dataset. Following the strategy used for C vulnerabilities and bugs, we collected and curated 61 confirmed Java vulnerability-fixing commit pairs with their 204 functions. This evaluation allows us to assess Mystique's ability to generalize across different programming languages. To adapt Mystique with Java vulnerability, we replaced Clang-tidy with a popular Java syntactic error checking tool [33].

Cross-LLM Porting Result. As shown in Table 7, when comparing Mystique-StarCoder with PPatHF-StarCoder, E-M increases from 0.365 to 0.689 at the CVE level and from 0.535 to 0.795 at the function level, while Succ rises from 0.385 to 0.710 at the CVE level and from 0.550 to 0.806 at the function level. The failure rate is significantly reduced from 0.450 to 0.194, demonstrating that Mystique generates fewer incorrect patches. These findings confirm that Mystique's methodology, including slicing, fine-tuning strategies, and iterative refinement, greatly enhances patch porting performance, independent of the underlying LLM. When replacing StarCoder with CodeLlama, both PPatHF and Mystique show further performance improvements. For PPatHF, switching to CodeLlama increases Succ by 0.135 at the CVE level and 0.120 at the function level. For Mystique, using CodeLlama leads to a more substantial boost, with Succ increasing by 0.214 at the CVE level and 0.148 at the function level. These results highlight that CodeLlama is a superior LLM for patch porting compared to StarCoder, but the most significant gains come from the combination of Mystique's methodology with CodeLlama, demonstrating that both the methodology and the LLM choice contribute to achieving the state-of-the-art performance.

Cross-Project Porting Result. The results shown in Table 8 highlight the superior generalization capabilities of Mystique compared to PPatHF across different projects. In the L./O. group, Mystique achieved 150 (0.829) E-M and 152 (0.840) Succ at CVE level and 286 (0.888) E-M and 288 (0.894) Succ at function level, with a C-B of 0.878, respectively. In contrast, PPatHF achieved only 42 (0.232) E-M and 44 (0.243) Succ at CVE level, 110 (0.342) E-M and 115 (0.357) Succ at function level, and a C-B of 0.749, respectively. Mystique outperforms PPatHF by 0.597 (245.7%) and 0.537 (150.4%) in Succ at CVE and function level in the L./O. group. Similarly, in the O./L. group, Mystique maintained robust performance with 470 (0.916) and 985 (0.950) Succ at CVE and function level, respectively. Meanwhile, PPatHF again exhibited a notable decline, achieving only 131 (0.255) and

Table 9. Results of Our Generalization Evaluation Across Bugs

Tool	# Bug (Func.)	Success (Bug Level)		Success (Func. Level)		Failure (Func. Level)	
		E-M (Δ E-M)	Succ (Δ Succ)	E-M (Δ E-M)	Succ (Δ Succ)	Fail	C-B
FixMorph (L.)	185 (340)	35 (0.189)	38 (0.205)	69 (0.203)	75 (0.221)	265 (0.779)	0.638
Mystique (L.)		**164 (0.886)**	**164 (0.886)**	**317 (0.932)**	**317 (0.932)**	**23 (0.068)**	**0.891**
TSBPORT	652 (1,185)	489 (0.750)	494 (0.758)	920 (0.776)	932 (0.786)	253 (0.214)	0.853
GPT-3.5		187 (0.287)	198 (0.304)	413 (0.349)	430 (0.363)	755 (0.637)	0.698
GPT-4o		420 (0.644)	431 (0.661)	903 (0.762)	918 (0.775)	267 (0.225)	0.709
CodeLlama		298 (0.457)	304 (0.466)	667 (0.563)	675 (0.570)	510 (0.430)	0.698
StarCoder		351 (0.538)	358 (0.549)	784 (0.662)	797 (0.673)	388 (0.327)	0.685
PPatHF		292 (0.448)	297 (0.456)	692 (0.584)	698 (0.589)	487 (0.411)	0.738
Mystique		**576 (0.883)**	**577 (0.885)**	**1,103 (0.931)**	**1,104 (0.932)**	**81 (0.068)**	**0.897**

464 (0.447) Succ at CVE and function level. Mystique outperforms PPatHF by 0.661 (259.2%) and 0.503 (112.5%) in Succ at CVE and function level in the O./L. group.

Bug Porting Result. The results shown in Table 9 demonstrate the consistent performance of Mystique, particularly in comparison to other tools. Mystique achieved 576 (0.883) E-M and 577 (0.885) Succ at bug level and 1,103 (0.931) E-M and 1,104 (0.932) Succ at function level. In contrast, PPatHF showed significantly lower performance, with 292 (0.448) E-M and 297 (0.456) Succ at bug level, and 692 (0.584) E-M and 698 (0.589) Succ at function level. TSBPORT is recognized as the best state-of-the-art with 489 (0.750) E-M and 494 (0.758) Succ at bug level, and 920 (0.776) E-M and 932 (0.786) Succ at function level. However, it experienced a performance drop in E-M and Succ compared with vulnerability patch porting. Mystique outperforms the best state-of-the-art TSBPORT by 0.127 (16.8%) and 0.146 (18.6%) at Succ rate at bug and function level. Other tools showed varied performance, with GPT-4o and StarCoder performing relatively better but still falling short of Mystique's consistency at both bug and function level.

Java Vulnerability Porting Result. The results shown in Table 10 show that Mystique achieved 51 (0.836) E-M and 52 (0.852) Succ at CVE level, and 192 (0.941) E-M and 194 (0.951) Succ at function level. Mystique outperformed the best approaches GPT-4o in adapting to a different programming language by 0.229 (37.7%) at E-M and 0.213 (33.3%) at Succ at CVE level, and 0.137 (17.0%) at E-M and 0.127 (15.4%) at Succ at function level, respectively. Pattern-based tools like TSBPORT and FixMorph are not compatible with other languages except C, with no results reported.

Summary: The overall evaluation demonstrates the superior generalization capabilities of Mystique across LLMs, projects, bugs, and programming languages. In the cross-LLM porting evaluation, comparing Mystique-StarCoder with PPatHF-StarCoder, Succ rises from 0.385 to 0.710 at the CVE level and from 0.550 to 0.806 at the function level. For Mystique, using CodeLlama leads to a more substantial boost, with Succ increasing by 0.214 at the CVE level and 0.148 at the function level. In the cross-project porting evaluation, Mystique outperforms PPatHF by 0.597 (245.7%) and 0.537 (150.4%) at Succ rate at CVE and function level in the L./O. group. Similarly, Mystique outperforms PPatHF by 0.661 (259.2%) and 0.503 (112.5%) at Succ rate at CVE and function level in the O./L. group. In the bug porting evaluation, Mystique outperforms the best state-of-the-art TSBPORT by 0.127 (16.8%) and 0.146 (18.6%) at Succ rate at bug and function level. In the Java vulnerability porting evaluation, Mystique demonstrates its adaptability, achieving 51 (0.836) E-M and 52 (0.852) Succ at CVE level, and 192 (0.941) E-M and 194 (0.951) Succ at function level, which outperforms GPT-4o by 0.213 (33.3%) and 0.127 (15.4%) at Succ rate at CVE and function level.

Table 10. Results of Our Generalization Evaluation Across Java

Tools	# CVE (Func.)	Success (CVE Level)		Success (Func. Level)		Failure (Func. Level)	
		E-M (Δ E-M)	Succ (Δ Succ)	E-M (Δ E-M)	Succ (Δ Succ)	Fail	C-B
PPatHF	61 (204)	23 (0.377)	23 (0.377)	131 (0.642)	131 (0.642)	73 (0.358)	0.566
GPT-3.5		16 (0.262)	18 (0.295)	110 (0.539)	112 (0.549)	92 (0.451)	0.702
GPT-4o		37 (0.607)	39 (0.639)	164 (0.804)	168 (0.824)	36 (0.176)	0.787
CodeLlama		30 (0.492)	30 (0.492)	136 (0.667)	136 (0.667)	68 (0.333)	0.695
StarCoder		35 (0.574)	36 (0.590)	161 (0.789)	162 (0.794)	42 (0.206)	0.641
Mystique		**51 (0.836)**	**52 (0.852)**	**192 (0.941)**	**194 (0.951)**	**10 (0.049)**	**0.830**

Table 11. Results of Our Efficiency Evaluation

Time (s)	Mystique	TSBPORT	FixMorph	PPatHF	GPT-3.5	GPT-4o	CodeLlama	StarCoder
Per CVE	110.8	123.9	3,239.8	69.4	29.8	**24.9**	64.3	55.7
Per Func.	56.5	63.2	1,603.7	35.4	15.2	**12.7**	32.8	28.4

4.6 Efficiency Evaluation (RQ5)

RQ5 Setup. We measured the average time taken to port patch for each CVE as well as for each function. The fine-tuning took 5.1 hours for Mystique and 19.4 hours for PPatHF.

Overall Result. As shown in Table 11, Mystique takes 56.5 seconds per function and 110.8 seconds per CVE to port patches, closely matching the performance of PPatHF, which takes 35.4 seconds per function and 69.4 seconds per CVE. Mystique significantly outperforms pattern-based approaches FixMorph and TSBPORT. While GPT-3.5 and GPT-4o are faster, completing patch porting in just 10 to 30 seconds. The time cost of Mystique owes to two parts, i.e., signature generation and refining. Specifically, for each CVE, Mystique spends 75.3 seconds on signature generation by Joern and 35.5 seconds on refining, leading to a total of 110.8 seconds. For each function, Mystique spends 38.4 seconds on signature generation and 18.1 seconds on refining, totaling 56.5 seconds. We believe that they are acceptable given Mystique's superior effectiveness for the patch porting task.

Summary: Mystique takes 56.5 and 110.8 seconds per function and CVE to port patches.

4.7 Usefulness Evaluation (RQ6)

RQ6 Setup. We first extracted all C/C++ and Java vulnerabilities from CVE/NVD, along with their referenced patches, from January 1, 2020, to September 1, 2024. After extraction, we collected 810 C-related CVEs and 368 Java-related CVEs. Next, we employed three state-of-the-art vulnerability code clone detection tools (i.e., DeepDFA [45], FIRE [10], and MVP [58]) to identify vulnerabilities that remained vulnerable across the original repository branches and related forked branches. To ensure the presence of vulnerabilities while minimizing human effort, we extracted the intersection of vulnerabilities identified by the three approaches across the branches. After manual verification, we identified a total of 16 CVEs, which included 10 CVEs with 24 vulnerable branches in C repositories and 6 CVEs with 10 vulnerable branches in Java repositories.

Overall Result. Table 12 presents the results of our usefulness evaluation. Mystique performed automated patch porting for these CVEs and successfully ported 34 patches across the branches. We then reported the porting results through 34 pull requests (PRs) to the repository developers. Developers have accepted 29 PRs with the ported patches by Mystique. However, 5 PRs have not been merged in a timely manner. Among these, 1 PR was not merged because the vulnerability was considered low severity, despite the ported patch was confirmed correct by the developers. 4 PRs are still awaiting confirmation from developers.

Table 12. Pull Request Results of Our Usefulness Evaluation

CVE	Language	Patched Repository@Branch	# Vul. Branches	Merge Status
CVE-2024-41565	Java	mezz/JustEnoughItems@1.21.x	1	✓
CVE-2024-26579	Java	apache/inlong@master	1	✓
CVE-2020-1926	Java	apache/hive@branch-2.3	6	✓
CVE-2021-3878	Java	stanfordnlp/CoreNLP@main	1	✓
CVE-2024-43406	Java	lf-edge/ekuiper@master	1	✓
CVE-2022-47021	C/C++	xiph/opusfile@master	2	✓
CVE-2024-6381	C/C++	mongodb/mongo-c-driver@r1.26	2	✓
CVE-2021-43814	C/C++	rizinorg/rizin@dev	1	✓
CVE-2023-27590	C/C++	rizinorg/rizin@dev	2	✓
CVE-2020-24370	C/C++	lua/lua@master	9	✓
CVE-2020-14147	C/C++	redis/redis@unstable	4	✓
CVE-2024-34696	Java	geoserver/geoserver@main	1	△
CVE-2023-28097	C/C++	OpenSIPS/opensips@master	1	△
CVE-2023-32690	C/C++	NVIDIA/open-gpu-kernel-modules@550	1	△
CVE-2024-39894	C/C++	freebsd/freebsd-src@main	1	△
CVE-2024-28882	C/C++	OpenVPN/openvpn@master	1	✗

Summary: MYSTIQUE successfully ported patches of 16 CVEs in C and Java repositories, and submitted 34 PRs to those repositories. 29 of the 34 PRs have been successfully merged into the repository branches or related forked branches.

4.8 Discussion

Threats. First, our human evaluation to determine the correctness of generated patch relies on the patches submitted by developers. However, some developers' patches may only partially fix the issues or include errors, which introduces potential bias into our human evaluation process. Second, the configurable parameters in MYSTIQUE, such as those for slicing, refining and LLM, are empirically set. To mitigate this threat, we evaluate the robustness of MYSTIQUE by assessing its effectiveness through parameter sensitivity analysis. Third, open-source LLMs are advancing rapidly, with more sophisticated models like StarCoder2 [2] emerging regularly. We plan to integrate MYSTIQUE with these advanced code LLMs to explore its generalizability across different underlying LLMs. Last, the results of usefulness evaluation largely depend on the effectiveness of the vulnerability clone detection approaches. Some vulnerable branches may go undetected by these tools, which establishes the lower bound of MYSTIQUE's overall usefulness of patch porting in practice.

Limitations. First, a primary limitation of MYSTIQUE is that it does not consider inter-function semantic and syntactic features, such as function calls, which limits its ability to port patches for inter-procedural vulnerabilities. Second, the semantic function level slicing does not incorporate knowledge beyond the function itself, such as the structure of C code. Last, the accuracy of slicing depends on Joern. While we have addressed some of Joern's buggy logic to improve patch porting performance, there may still be limitations related to its implementation.

5 Related Work

Patch Porting. Several studies have highlighted the high rate of vulnerability inheritance and delays in porting patch within open-source projects after branching [6, 35, 41, 42]. These delays leave open-source projects exposed to security risks, and adapting patches across different branches is often error-prone for developers [3, 40, 47]. To assist developers in managing patch porting, several automated approaches (e.g., pattern-based and LLM-based approaches) have been proposed. FIXMORPH [41] is a syntactical pattern-based approach that leverages clang/llvm to port patch to

lower versions. TSBPORT [62] is a semantical pattern-based approach that fixes vulnerabilities. Pattern-based approaches are limited by their dependence on predefined patterns, which hinders their generalization when modifications fall outside these patterns. Moreover, these approaches struggle with function level changes that require a nuanced semantic understanding, extending beyond local hunks. PPatHF [35], an LLM-based approach, shows potential in patch porting by leveraging fine-tuned LLM. However, this approach simplifies the patching process by reducing compound blocks without any changed statement to focus on core statements, which can inadvertently strip away critical syntactic or semantic information. Additionally, its heavy reliance on intra-repository knowledge for LLM fine-tuning can lead to misalignments with actual porting tasks, especially when LLM is used to port patches in projects that were not included during the fine-tuning stage, without any further refinement. SkyPort [43] defines a list of sink functions for PHP injection vulnerabilities, limiting their applicability to other programming languages or vulnerability types. PatchWeave [42] utilizes exploitable test cases to guide patch porting, but its effectiveness is also constrained by the availability of such test cases.

Automated Program Repair (APR). APR is a related but different problem from patch porting. After the fundamental APR approaches of GenProg [19, 53], there are five mainstream APR approaches, which are heuristic-based, constraint-based, pattern-based, DL-based, and very recent LLM-based. Heuristic-based approaches (e.g., [11, 19, 51, 53, 64–66]) use the genetic algorithms to generate patch. Some improved approaches (e.g., [17, 51, 59]) narrow down the searching space by using the similar code in code databases. Pattern-based approaches (e.g., [1, 18, 21]) use pre-defined fix patterns that are summarized by human, which faces the same problems with the pattern-based porting approaches. Constraint-based approaches (e.g., [8, 16, 24, 25, 61]) usually focus on a single conditional expression and employ advanced constraint-solving or synthesis techniques to synthesize candidate patches. DL-based approaches (e.g., [4, 57, 63]) or very recent LLM-based approaches (e.g., [9, 13, 15, 50, 52, 56]) aim to automate patch generation by employing DL models that can capture the semantic features of programs and generate candidate patches. All APR approaches take buggy code and test suites as inputs to generate candidate patches, which do not fit the patch porting task where the inputs are an original patch and a target vulnerable function.

6 Conclusions

We have developed Mystique, a novel approach for automatically porting security patches. Mystique takes both the patch and vulnerable functions as inputs, extracts semantic and syntactic vulnerability-relevant signatures from the patch, and identifies corresponding signature in the vulnerable function. Using a fine-tuned LLM, Mystique ports the patch and iteratively refines it for higher success rate of patch porting. Extensive experiments have demonstrated Mystique's effectiveness, generality, and practical usefulness. In future work, we plan to integrate additional LLMs to further enhance Mystique's patch porting capabilities and support more languages.

7 Data Availability

The source code for Mystique, with experimental data and results, is available at our website [48].

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

References

[1] Afsoon Afzal, Manish Motwani, Kathryn T Stolee, Yuriy Brun, and Claire Le Goues. 2019. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2162–2181.

[2] bigcode. 2024. StarCoder2-15B. https://huggingface.co/bigcode/starcoder2-15b.

[3] Debasish Chakroborti, Chanchal Roy, and Kevin Schneider. 2024. A Study of Backporting Code in Open-Source Software for Characterizing Changesets. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 296–297.

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

[5] Clang. 2024. *clang-tidy*. Retrieved August 25, 2024 from https://clang.llvm.org/extra/clang-tidy

[6] Alexandre Decan, Tom Mens, Ahmed Zerouali, and Coen De Roover. 2021. Back to the past–analysing backporting practices in package dependency networks. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4087–4099.

[7] Ekwa Duala-Ekoko and Martin P Robillard. 2007. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*. 158–167.

[8] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. 85–91.

[9] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering*. 1469–1481.

[10] Siyue Feng, Yueming Wu, Wenjie Xue, Sikui Pan, Deqing Zou, Yang Liu, and Hai Jin. 2024. {FIRE}: Combining {Multi-Stage} Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1867–1884.

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

[12] git. 2024. *git-cherry-pick*. Retrieved August 25, 2024 from https://git-scm.com/docs/git-cherry-pick

[13] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1471–1493.

[14] J. E. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *ArXiv* abs/2106.09685 (2021).

[15] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. 1162–1174.

[16] Zunchen Huang and Chao Wang. 2024. Constraint Based Program Repair for Persistent Memory Bugs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.

[18] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. 295–306.

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

[20] Xingyu Li, Zheng Zhang, Zhiyun Qian, Trent Jaeger, and Chengyu Song. 2024. An Investigation of Patch Porting Practices of the Linux Kernel Ecosystem. In *Proceedings of the 2024 IEEE/ACM 21st International Conference on Mining Software Repositories*. 63–74.

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

[22] I Loshchilov. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[23] Arthur Roberto Marcondes and Ricardo Terra. 2020. An approach for updating forks against the original project. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. 213–222.

[24] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Proceedings of the 10th International Symposium of Search-Based Software Engineering*. 65–86.

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[26] Meta. 2024. CodeLlama-13b-Instruct. https://huggingface.co/meta-llama/CodeLlama-13b-Instruct-hf.

[27] NVD. 2023. *NVD*. Retrieved July 14, 2023 from https://nvd.nist.gov/vuln/data-feeds

[28] NVD. 2024. *CVE-2023-0465*. Retrieved August 25, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2023-0465

[29] NVD. 2024. *CVE-2023-1994*. Retrieved August 25, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2023-1994

[30] OpenAI. 2024. GPT-3.5 Turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.
[31] OpenAI. 2024. GPT-4o. https://platform.openai.com/docs/models/gpt-4o.
[32] openssl. 2024. *GitHub repository of openssl.* Retrieved August 25, 2024 from https://github.com/openssl/openssl
[33] oracle. 2024. *Interface JavaCompiler.* Retrieved August 25, 2024 from https://docs.oracle.com/javase/8/docs/api/javax/tools/JavaCompiler.html
[34] Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shanping Li. 2023. Fine-grained commit-level vulnerability type prediction by CWE tree structure. In *Proceedings of the 45th International Conference on Software Engineering*. 957–969.
[35] Shengyi Pan, You Wang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2024. Automating Zero-Shot Patch Porting for Hard Forks. *arXiv preprint arXiv:2404.17964* (2024).
[36] Shaun Phillips, Jonathan Sillito, and Rob Walker. 2011. Branching and merging: an investigation into current version control practices. In *Proceedings of the 4th international workshop on cooperative and human aspects of software engineering*. 9–15.
[37] Sebastian Raschka. 2024. *Finetuning LLMs with LoRA and QLoRA: Insights from Hundreds of Experiments.* Retrieved August 25, 2024 from https://lightning.ai/pages/community/lora-insights
[38] Baishakhi Ray, Christopher Wiley, and Miryung Kim. 2012. REPERTOIRE: a cross-system porting analysis tool for forked software projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–4.
[39] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
[40] Luis R Rodriguez and Julia Lawall. 2015. Increasing automation in the backporting of Linux drivers using Coccinelle. In *Proceedings of the 2015 11th European Dependable Computing Conference*. 132–143.
[41] Ridwan Shariffdeen, Xiang Gao, Gregory J Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated patch backporting in Linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 633–645.
[42] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2020. Automated patch transplantation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–36.
[43] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches. In *Proceedings of the 31st USENIX Security Symposium*. 1993–2010.
[44] ShiftLeftSecurity. 2024. *Joern.* Retrieved April 20, 2024 from https://github.com/ShiftLeftSecurity/joern
[45] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[46] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. 2022. Understanding the practice of security patch management across multiple branches in oss projects. In *Proceedings of the ACM Web Conference 2022*. 767–777.
[47] Ferdian Thung, Xuan-Bach D Le, David Lo, and Julia Lawall. 2016. Recommending code changes for automatic backporting of Linux device drivers. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution*. 222–232.
[48] Mystique. 2024. *Mystique.* Retrieved May 25, 2024 from https://doi.org/10.5281/zenodo.14840986
[49] tree sitter. 2023. Tree-sitter: An incremental parsing system for programming tools. Retrieved September 1, 2024 from https://tree-sitter.github.io/tree-sitter/
[50] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.
[51] Yingyi Wang, Yuting Chen, Beijun Shen, and Hao Zhong. 2017. Crsearcher: Searching code database for repairing bugs. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*. 1–6.
[52] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. 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*. 172–184.
[53] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. 364–374.
[54] wiki. 2024. *Levenshtein Distance.* Retrieved May 25, 2024 from https://en.wikipedia.org/wiki/Levenshtein_distance
[55] Wireshark. 2024. *GitHub repository of Wireshark.* Retrieved August 25, 2024 from https://github.com/wireshark/wireshark
[56] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering*. 1482–1494.

[57] Chunqiu Steven Xia and Lingming Zhang. 2022. 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*. 959–971.

[58] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium*. 1165–1182.

[59] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. 660–670.

[60] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[61] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[62] Su Yang, Yang Xiao, Zhengzi Xu, Chengyi Sun, Chen Ji, and Yuqing Zhang. 2023. Enhancing OSS Patch Backporting with Semantics. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2366–2380.

[63] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 678–690.

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

[65] Yuan Yuan and Wolfgang Banzhaf. 2019. A hybrid evolutionary system for automatic software repair. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1417–1425.

[66] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward better evolutionary program repair: An integrated approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–53.

Received 2024-09-13; accepted 2025-01-14