Lares: LLM-driven Code Slice Semantic Search for Patch Presence Testing

Siyuan Li[1,2,3], Yaowen Zheng[2,3], Hong Li[2,3,*], Jingdong Guo[2,3], Chaopeng Dong[2,3]
Chunpeng Yan[2,3], Weijie Wang[2,3], Yimo Ren[2,3], Limin Sun[2,3], Hongsong Zhu[2,3]
[1]*School of Cyber Science and Technology, Shandong University, Shandong, China*
[2]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[3]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*

siyuan@sdu.edu.cn, {zhengyaowen,lihong,guojingdong,dongchaopeng,yanchunpeng}@iie.ac.cn,
{wangweijie,renyimo,sunlimin,zhuhongsong}@iie.ac.cn

***Abstract*—In modern software ecosystems, 1-day vulnerabilities pose significant security risks due to extensive code reuse. Identifying vulnerable functions in target binaries alone is insufficient; it is also crucial to determine whether these functions have been patched. Existing methods, however, suffer from limited usability and accuracy. They often depend on the compilation process to extract features, requiring substantial manual effort and failing for certain software. Moreover, they cannot reliably differentiate between code changes caused by patches or compilation variations.**

To overcome these limitations, we propose Lares, a scalable and accurate method for patch presence testing. Lares introduces Code Slice Semantic Search, which directly extracts features from the patch source code and identifies semantically equivalent code slices in the pseudocode of the target binary. By eliminating the need for the compilation process, Lares improves usability, while leveraging large language models (LLMs) for code analysis and SMT solvers for logical reasoning to enhance accuracy. Experimental results show that Lares achieves superior precision, recall, and usability. Furthermore, it is the first work to evaluate patch presence testing across optimization levels, architectures, and compilers. The datasets and source code used in this article are available at https://github.com/Siyuan-Li201/Lares.

***Index Terms*—Patch Presence Testing, Binary Analysis, Large Language Model.**

I. Introduction

Software development is not a repetitive process. Developers often reuse third-party libraries to accelerate development, resulting in a complex software supply chain ecosystem [1]. However, this practice introduces security risks by propagating vulnerabilities. According to the Synopsys report [2], 96% of tested software reused at least one third-party library, with 89% of codebases containing open-source components that had not been updated for over two years, and some for more than four years. The failure to update and patch vulnerabilities promptly exacerbates the impact of 1-day vulnerabilities.

Several approaches [1], [3]–[8] have been proposed for detecting 1-day vulnerabilities in software. These approaches typically collect third-party libraries and vulnerable functions, performing function- or library-level matching using binary similarity techniques. However, they primarily identify functions similar to vulnerable ones, struggling to differentiate between vulnerable and patched functions due to minimal modifications in patches [9]. To address this limitation, patch presence testing was introduced, first proposed by Fiber in 2018 [10]. As in Figure 1, this technique analyzes a target function and patch information (e.g., vulnerable and patched versions) to determine whether the target function is closer to the vulnerable or patched version, enabling fine-grained distinction. We classify patch presence testing methods into two categories: syntactic-based and semantic-based approaches.

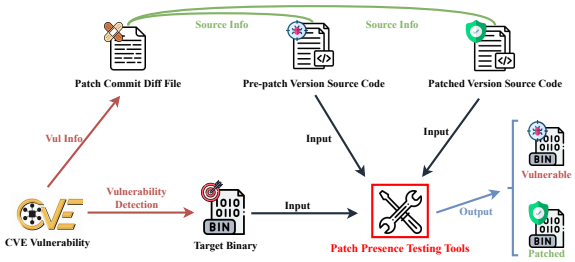

Fig. 1. The Background.

Syntactic-based methods usually use syntactic features for detection. BinXray [11] uses the sequence of mnemonic operators in binary instructions, function calls and constant values for detection, and combines structural statistics such as the number of instructions, basic blocks, branches and the control flow graph. Fiber [10] extracts the differences in control flow graphs and abstract syntax trees from the vulnerable version and patch version of the kernel to detect patches in the target kernel. PatchDiscovery [12] uses basic blocks with more changed instructions as representative blocks to improve the performance of BinXray [11].

Semantic-based methods usually use symbolic execution to extract various types of semantic features. PDiff [13] first locates the anchor block in CFG, and then uses symbolic execution to obtain semantic features such as path constraints to reach the anchor block for detection. Robin [9] generates

* corresponding author: lihong@iie.ac.cn

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00081

a malware function input (MFI) through symbolic execution from reference binaries, and collects four runtime features to compare whether the target binary is closer to the vulnerability version or the patch version. PS3 [14] also uses symbolic simulation to extract four similar semantic features (write data to reg, Store data to mem, Condition, call&Return) for comparison.

However, it is essential to note that existing methods exhibit certain limitations, which hinder their performance and usability in the real world. We summarize the limitations of the existing works in the following three points.

Firstly, existing methods rely on compilation to extract features, which is often difficult to automate and may fail for many software projects, reducing usability (**P1**). Patch information resides at the source code level, while the detection target is a binary file. Current approaches compile both vulnerable and patched versions into binaries [9], [11], [14] or intermediate representations (IR) [15] to extract features using debugging information. As a result, these methods rely on manual compilation, making them unsuitable for large-scale deployment. To address these limitations, we propose to eliminate the need for compilation, directly extracts features from patches and source code, and enables efficient, automated large-scale detection.

Secondly, existing methods struggle to locate patch-related code fragments accurately, reducing detection accuracy (**P2**). Binary similarity matching typically operates at the function or library level, whereas patch detection requires finer-grained comparison of specific code fragments. While extracting patch-related fragments from reference binaries (e.g., vulnerable and patched functions) is straightforward, identifying corresponding fragments in target binaries remains challenging. Existing methods [9], [14] tend to directly match all features of the entire function with patch features. An effective method is needed to precisely locate and extract the corresponding code fragments from target binaries.

Thirdly, existing methods struggle to maintain accuracy across different compilation environments (**P3**). In real-world scenarios, target binaries may be compiled using varying optimization options, compilers, or architectures, making it difficult to distinguish between changes introduced by patches and those caused by compilation differences. Existing approaches partially mitigate the impact of optimization options but still suffer significant accuracy degradation. Furthermore, most existing approaches [9], [11], [14] only support feature comparison within the same architecture and do not support cross-architecture detection. In addition, they need to write feature extraction code specifically for each new architecture. Only REACT claims to support cross-architecture detection from the IR level, but it has not been evaluated on a cross-architecture dataset.

To solve the problems mentioned above, we propose Lares, an LLM-driven code slice semantic search technique that enhances the usability and accuracy of binary patch presence testing. Lares identifies the code slice corresponding to the patch source code from decompiled pseudocode and compares their semantic differences. Unlike existing methods, Lares does not require a compilation process and instead extracts features directly from the patch diff and patch function source code (**for P1**). It uses the decompiled pseudocode of the target binary function for comparison, with features that are architecture-independent and applicable to stripped binaries (**for P3**). Additionally, inspired by prior work on LLMs in decompilation [16] and function summarization [17], Lares employs LLMs to locate patch-related fragments, ensuring consistent granularity for precise comparison (**for P2**).

Lares comprises three modules. First, the patch enhancement module expands patch semantics by adding context-related statements to patch code with few statements. Second, the patch localization module leverages an LLM to identify pseudocode slices corresponding to patch code slices within the target function's pseudocode. Finally, the patch verification module validates the localization results to determine whether the target corresponds to a vulnerability or a patch by LLM and SMT Solver. The core insight of Lares lies in utilizing the code analysis and logical reasoning capabilities of LLMs to locate patch-related statements in pseudocode and compare them with source code slices from both the pre-patch and patched versions.

We thoroughly analyze the issues introduced by the compilation process in existing methods and demonstrate how Lares naturally avoids them. Our evaluation shows that Lares's lightweight design not only maintains usability but also improves precision and recall by effectively addressing P2 and P3. Compared to existing methods, Lares achieves a significant F1 score improvement of 9%-10%. This balance of accuracy and usability makes Lares a more effective solution for patch presence testing.

We summarize our main contributions below:

- We propose Lares, the first compile-free, LLM-based approach for patch presence testing tasks, which eliminates substantial manual work and achieves large-scale automated detection.
- We implement Lares with 3,000 lines of Python code and integrate it with several source and binary analysis tools, while also validating its performance using advanced LLMs.
- We conduct an extensive evaluation of Lares, and the results show that Lares outperforms state-of-the-art approaches in terms of both accuracy and recall overall across a range of architectures and compilation options.

II. Motivation

We illustrate our motivation and insights with the example shown in Figure 2. First, we demonstrate the necessity of conducting patch presence testing. Next, we analyze the limitations of existing approaches and present our insights for developing more robust tools for patch presence testing.

A. Necessity of Patch Presence Testing

Figure 2 illustrates CVE-2013-6449 as a representative example. The first box on the left contains the patch commit

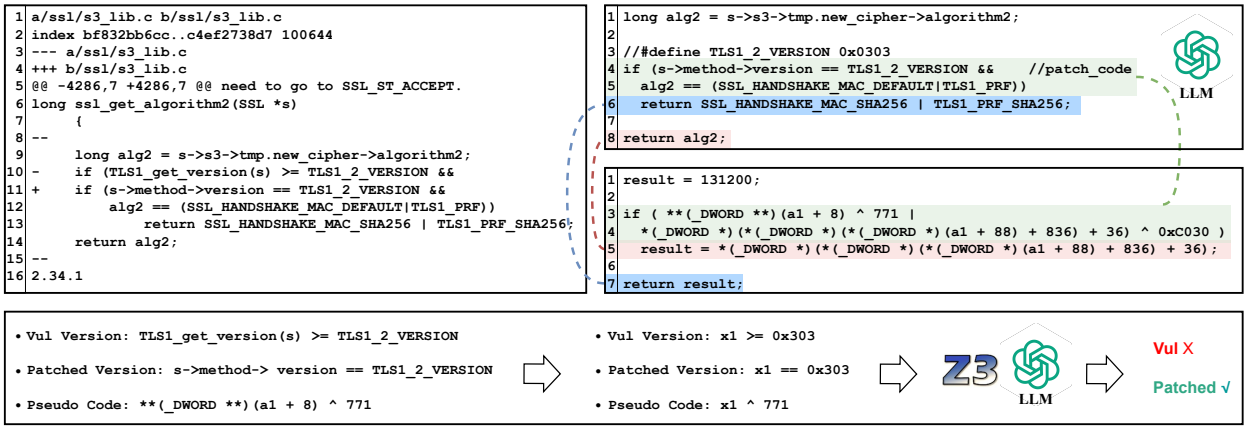

Fig. 2. A motivation example.

for the vulnerability, which introduces minimal code changes. Specifically, it replaces a version-related macro definition with a structure pointer and modifies the comparison operator from ">=" to "==". These subtle changes render the patched function highly similar to the vulnerable one. Consequently, existing binary similarity detection methods struggle to differentiate between the vulnerable and patched versions, as they primarily identify function-level similarities. This highlights the need for a more fine-grained patch existence verification method to detect the presence of the vulnerability accurately.

B. Limitations and Insights

1) Reliance on Project Compilation: Existing methods rely heavily on the compilation process to extract features, such as control flow graphs (CFG) [11], assembly code [9], [14], or intermediate representations (IR) [15]. For each vulnerability to be detected, these methods require manually compiling both pre-patch and patched binaries. This process necessitates the inclusion of debugging information to identify the precise location of the patch code within the binary. While B2SFinder [18] attempted to automate the compilation process, they reported that only 25% of the GitHub projects they tested could be successfully compiled automatically. Furthermore, we observed that compilation is not only challenging to fully automate but often requires extensive domain knowledge, with many projects failing even under manual compilation. These limitations render existing methods insufficient for real-world applications. Table I describes the specific problems caused by the compilation process.

External Dependencies: Some projects require resolving external library dependencies during compilation (e.g., Zlib for OpenSSL, Libpcap for Tcpdump), but they do not provide an automatic way to build the dependencies. This demands domain knowledge and careful alignment of compilation parameters (e.g., the path of external libraries compiled manually), hindering automation and increasing the likelihood of failures in complex projects.

Compilation Parameters: Reference binaries must be compiled with specific parameters (e.g., -O0, -g). Manually configuring compiler and linker options in the project-specific config file further reduces usability.

TABLE I
USABILITY EVALUATION OF EXISTING METHODS

Problems	REACT	BinXray	Robin	PS3	Lares
ED	✘	✘	✘	✘	✔
CP	✘	✘	✘	✘	✔
CV	✘	✘	✘	✘	✔
BT	✘	✘	✘	✘	✔
AC	✘	✔	✔	✔	✔

Note: ED: External Dependencies. CP: Compilation Parameters. CV: Compiler Versions. BT: Build Tools. AC: Additional Compilation. ✔ indicates that this method can solve the problem, while ✘ indicates that it cannot.

Compiler Versions: Variations in compiler versions can cause compatibility issues. Older projects may fail with newer compilers due to stricter checks, while newer projects may rely on C++20 features supported only in g++ v10 or later.

Build Tools: Projects use diverse build tools (e.g., Automake, CMake, Bazel), which complicates the setup process. Datasets from prior work [19] such as exjson even require MIX as the build tool.

Additional Compilation: Methods like REACT require extracting the compilation parameters during the compilation process and re-compiling to generate intermediate representations (IR) using Clang, further complicating the process and increasing the risk of errors.

Therefore, we aim to propose a **compile-free approach** for patch presence testing. Unlike existing methods, we decompile the target binary into pseudocode and analyze it by extracting features from the source code for comparison to determine the presence of a patch. This compile-free design inherently addresses all issues in the compilation process.

Avoiding compilation is a critical advantage. In practical application scenarios, the analysis targets are often real-world binaries extracted from software or firmware. Although these projects may be technically compilable, the compilation

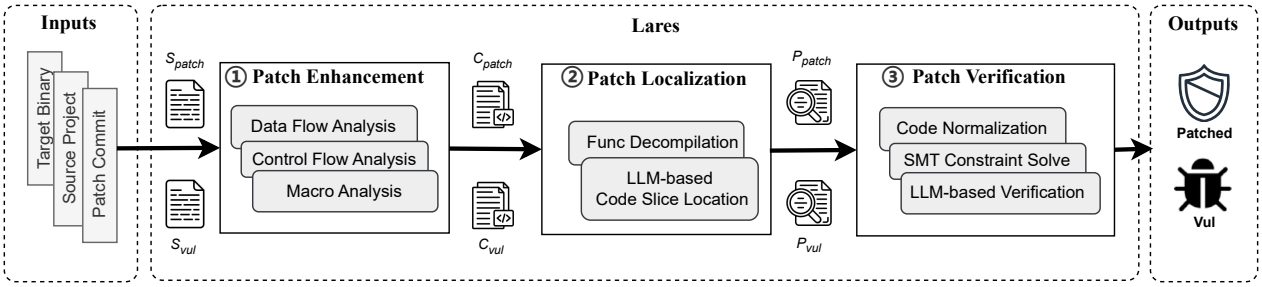

Fig. 3. The workflow of Lares.

process is difficult to automate and often requires manual, case-by-case configuration. For large-scale vulnerability analysis tasks that demand efficient processing across numerous projects, these issues significantly hinder system efficiency and scalability.

2) Inaccuracy in Patch Localization: In addition, existing methods, such as PS3 [14], extract features from the entire function code in the target binary and match these features with those of vulnerabilities or patches. However, this approach introduces numerous irrelevant features, as patch statements often constitute only a small portion of the function. Other methods, including BinXray [11], Robin [9], and PDiff [13], rely on control flow graph (CFG) blocks and heuristic rules to locate anchor points. However, CFG blocks are sensitive to changes in the compilation environment, leading to inaccuracies in these methods. To address the performance degradation caused by inconsistent comparison units, we aim to locate the code slices in the target function that correspond to the patch code slices. This method should be robust to compilation environment variations and leverage semantic analysis to identify the relevant code slices for comparison.

Therefore, we aim to propose an approach that can **accurately locate patch-related code within the target** and perform subsequent analysis and detection. As shown in the two boxes in the upper corner of Figure 2, we directly compare the patch-related source code with the pseudocode obtained through decompilation of the target binary. Specifically, we leverage LLM to find pseudocode slices corresponding to patch-related source code from pseudocode, as highlighted in Figure 2. The dotted lines illustrate the correspondence between these statements. This patch localization enables comparison at a consistent and fine-grained code slice level.

However, LLMs cannot perfectly locate the patch or vulnerability slice. While LLMs can identify the pseudocode slice most similar to the target patch fragment, the retrieved slice may precede or follow the actual patch. Therefore, an additional patch verification step is necessary. Similar to prior work [9], [14], [15], we employ an SMT solver for semantic-level comparison. As shown in the bottom box of Figure 2, we extract key instructions from the code fragment and normalize them into unified equations. Using the Z3 solver, we evaluate semantic equivalence, effectively handling cases such as $x1\hat{}771$ and x1 == 0x303, which differ syntactically but are semantically equivalent. For cases where the Z3 solver fails, we leverage LLMs to further analyze the match between the target code slice and the vulnerability or patch slice to determine the correct correspondence.

III. Methodology

In this section, we introduce the design of Lares. We begin by explaining its overall workflow, followed by a detailed description of each module.

A. Overview

The workflow of Lares, illustrated in Figure 3, consists of three phases: patch enhancement, patch localization, and patch verification. We assume that a binary code similarity detection scheme (e.g., BinaryAI [8]) has been used to identify potential vulnerable functions F_t in the target binary, and the goal of this work is to further verify whether the vulnerability has been patched. We take the target binary, source code project, and patch commit as input. Then, Lares determines whether the vulnerable code in the target binary has been patched.

Patch Enhancement. Given the source code of the vulnerable function S_{vul} and that of the patched function S_{patch} as input, this module aims to extract patch-related code slices for both versions (C_{vul} and C_{patch}) to allow subsequent patch localization within the target. Specially, some patches modify only a small portion of the source code, sometimes as little as a single line, making the code diff between the vulnerable and patched versions insufficient. Therefore, the goal is to extract more enriched code slices that include only statements relevant to the patch and avoid the introduction of irrelevant code.

Patch Localization. In this step, we first decompile F_t into pesudocode P_t. Next, we aim to locate the corresponding slice within P_t using C_{vul} and C_{patch} obtained from the previous step.

Currently, LLMs have demonstrated strong code analysis capabilities in tasks such as code decompilation [16] and function summarization [17]. Therefore, we designed an LLM-based approach to locate code slices within P_t. Specifically, we crafted a tailored prompt containing P_t, C_{vul} and C_{patch}. The LLM is tasked with mapping each line of the source code slice to its corresponding pseudocode sequence, and finally get the pseudocode slices (P_{vul} and P_{patch}) corresponding to C_{vul} and C_{patch} . While this result identifies the pseudocode slices

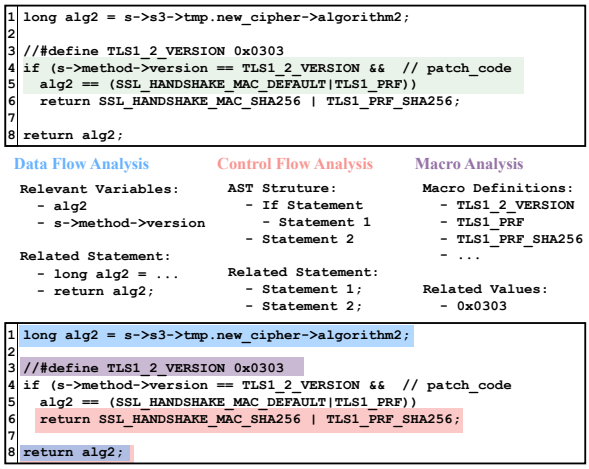

Fig. 4. The Patch Enhancement Module.

most semantically similar to the source code slice, further verification is required to determine whether the fragment represents the vulnerability or the patch.

Patch Verification. From the previous step, we obtained two pairs, (C_{vul}, P_{vul}) and (C_{patch}, P_{patch}), representing the source code slices before and after the patch, along with their potential corresponding pseudocode slices in the target. In this step, we analyze the semantic equivalence of each pair. The pair with the highest semantic equivalence similarity is considered the best match, allowing us to infer whether the target corresponds to the pre-patch or patched version.

B. Patch Enhancement

In this phase, we utilize three core strategies to enrich patch-related code slices, including data flow analysis, control flow Analysis, and macro analysis. Given a patch commit diff file and the pre-patch and patched versions of the function source code (S_{vul} and S_{patch}), the module outputs the patch-related code slices for both versions (C_{vul} and C_{patch}). Similar to existing work MVP [20], Lares leverages lightweight slicing implemented via Joern [21], enabling a compile-free patch presence test. This method is intra-function.

1) Data Flow Analysis: The first enhancement strategy targets the data dependencies of the patch. Variables defined or used within the patch often act as critical links between the patch and the rest of the program. However, since patches typically modify only a small portion of the code, the broader context of these variables often remains unexplored. To address this, we conduct systematic data flow analysis to capture the complete role of these variables within the program.

We begin by parsing the patch code to extract all defined variables and used variables. These variables represent key points of interaction between the patch and its surrounding code. For instance, as shown in Figure 4, a patch modifying a single `if` statement would involve the condition of the `if` clause and any variables referenced within its body, which may significantly interact with other parts of the program.

After identifying the relevant variables, we analyze the broader function or module containing the patch to locate additional statements that define or use these variables. For example, in Figure 4, lines 1 and 8 are data flow-related statements connected to the patch statement, providing additional context for analysis.

2) Control Flow Analysis: While data flow analysis focuses on variable interactions, control flow analysis examines the structural and logical organization of the code impacted by the patch. This analysis is crucial for understanding how the patch affects the program's execution paths, particularly in the presence of complex control structures such as loops and conditional branches. By incorporating these dependencies, control flow analysis offers a comprehensive understanding of the patch's impact on program execution.

For each patch, we determine the control flow structure in which it resides. This involves identifying the entry points of control structures, such as the condition in an `if` statement or the header of a loop. We further analyze the structure by extracting the first statement of each block and the statement immediately following the block. For example, if a patch modifies the body of an `if` statement, we also capture the corresponding `else` branch, if present, along with the statement following the `if-else` block.

3) Macro Analysis: In addition to data flow and control flow, the third enhancement strategy focuses on macro definitions, which are commonly used in source code to represent constants, expressions, or inline code fragments. Since the target binary is stripped of such definitions, we must identify and resolve the specific values of macro definitions in the patch to better align with the values found in the pseudocode.

After identifying the relevant macro definitions, we locate their corresponding values or code fragments in the source code. These values are then substituted into the patch, replacing macro placeholders with their concrete representations. For example, in Figure 4, the macro `TLS1_2_VERSION` resolves to the value `0x0303`, enabling correct matching in later stages (e.g., aligning 0x303 with 771 via macros). Although the direct search strategy carries some risk of error, this lightweight approach is effective in practice and requires no compilation.

C. Patch Localization

The purpose of the Patch Localization module is to identify the corresponding pseudocode slice (P_{vul} and P_{patch}) within the target function's pseudocode, using the enhanced patch code slice from the previous module (C_{vul} and C_{patch}). Rather than using only the patch code slice as input, we utilize the entire function's source code. To assist identification, we annotate each line of the patch code slice within the function source code by adding the comment "`//patch_code`".

1) Truncation of Function: To address the token limits of large language models (LLMs), we apply AST-guided truncation to retain essential contextual information while ensuring syntactic and semantic integrity.

For patch code. we use the first statement of the patch as an anchor, and expand upward and downward to include sur-

TABLE II
PROMPT FOR EACH TASK IN LARES

Task	Prompt Template
Patch Localization	Suppose you are a software reverse engineer with strong code analysis skills. You have the source code of a function and the pseudo code obtained through binary decompilation. Lines in the source code that end with "//patch_code" are patch codes. Can you identify the patch codes in the pseudo code that corresponds to the patch code? Must only output your findings as a JSON dictionary. - Output format:<json_format_sample> - Source code: <source_code> - Pseudocode: <pseudo_code>
Patch Verification	You are a software reverse engineer analyzing decompiled pseudo code. Your task is to determine whether the code is patched or pre-patch version by analyzing the reliability of matching results. Must only output your findings as a JSON dictionary. - INPUT: 1. Diff File: <patch_diff_label>2. patched version matches: <patch_result_json>3. pre-patch version matches: <vul_result_json> - ANALYSIS REQUIREMENTS: Evaluate each match in patched and pre-patch: Semantic correctness, Logic consistency, Context compatibility, Potential false matches. Compare quality of matches: Which version has more reliable matches, Which matches might be incorrect, Overall semantic alignment - RULES: Only one result (patched version or pre-patch version) corresponds to the correct version. Better semantic match determines the version

rounding lines of code. To ensure semantic integrity, truncation occurs at structural boundaries in the AST. For example, if the patch modifies a statement within a loop, the entire loop body is included. This process continues until the truncated code reaches a predefined token limit. The final patch code is annotated with the label "`patch code`" for subsequent processing. We adopt a similar strategy for the pseudocode of the target function. Rather than extracting a single statement and its context, we extract multiple code slices truncated at the AST structure boundaries and input them into Lares for subsequent processing.

2) Prompt Construction: After truncating both the patch code and pseudocode, we construct prompts for the LLM to perform the patch localization task. The prompt follows a carefully designed template, as shown in Table II. This template includes placeholders for the patch code and a single pseudocode segment, which are replaced with the truncated patch code and one segment of pseudocode, respectively. The prompt instructs the LLM to identify the pseudocode slice corresponding to the patch code slice.

The constructed prompt is sent to the LLM, which analyzes the patch code and pseudocode segment and returns a JSON structure specifying the location of the patch code slice within the pseudocode. This JSON output includes the matched pseudocode slice and its corresponding location. The result is a precise mapping between the enhanced patch code slice and the corresponding pseudocode slice, providing the foundation for the subsequent *patch verification* module.

3) Reverse-Matching Strategy for Add and Delete Patches: Not all vulnerabilities identified during the *patch localization* module result in two matching pseudocode slices. For edit-type patches, it is possible to match the unique code before and after the patch with the target pseudocode, yielding two distinct results for the following verification. However, for add-only and delete-only patches, there is typically a single match between the added/deleted code slice and the pseudocode. This makes it challenging to determine whether the resulting pseudocode slice aligns more closely with the pre-patch or post-patch version. To address this, we introduce a novel **reverse-matching strategy** to ensure two matching results are always obtained.

Generating two pseudocode slices is practical and effective. Our goal is to compare the pseudocode with the two source codes before and after the patch. Because the patch-induced differences are minimal, an LLM cannot reliably locate a single pre- or post-patch slice. Instead, for each vulnerability, Lares matches the most similar pseudocode slice to the pre-patch source and another to the post-patch source, then selects the correct pairing to ensure accurate localization.

For add-only patches, in addition to the P_{patch} that matches the added code slice (C_{patch}), the second matching result P_{vul} is generated by reverse matching the pseudocode slice P_{patch} with the source code of the pre-patch version S_{vul}. This determines whether the pseudocode slice originates from the newly added code or from existing lines in the pre-patch version. Similarly, for delete-only patches, the P_{vul} is matched with the S_{patch} to get the P_{patch} for verification.

D. Patch Verification

The purpose of the patch verification module is to determine whether the pseudocode slice (P_{vul} and P_{patch}) identified by the patch localization module truly corresponds to the vulnerable code or the patched code (C_{vul} and C_{patch}). While the patch localization module identifies the pseudocode slice most similar to the patch code, additional analysis is required to distinguish whether the match aligns with the vulnerable or patched version. The Patch Verification module takes as input two locating results for the target function: one from the pre-patch version and one from the patched version. Its output is a determination of which locating result is correct.

Existing methods tend to use the Z3 solver for equivalence judgment. However, since feature extraction is performed on source code and pseudocode, it is difficult to fully obtain semantic formulas, which sometimes causes Z3 to fail.

Therefore, we combined LLM with Z3 for verification. This verification process is fully automated and forms the final step in the pipeline, enabling accurate and efficient analysis of software patches.

1) Lexical Analysis and Normalization: In order to extract semantic formulas from code slices and perform SMT-Based constraint solving, we need to first normalize them. The process begins with a customized **lexical analysis** of the matched code slices, where the code is parsed into fundamental lexical units. Finally, we extract the normalized equation from the vocabulary unit.

We use a custom finite state machine (FSM) for lexical analysis. We take each line of code as input, provide it to the FSM character by character, and perform state transition according to each new character. This recursive definition ensures that the FSM processes the input string sequentially, one character at a time, maintaining a systematic and deterministic approach to lexical analysis.

After lexical analysis, we get the lexical units corresponding to the code. From these units, we extract four primary types of code slices:

- **Conditional Statements:** e.g., `if (x > 0)`, `while (y == 1)`.
- **Assignment Statements:** e.g., `x = a + b`, `y = 0`.
- **Return Statements:** e.g., `return z`.
- **Function Calls:** e.g., `foo(x, y)`, `bar(x+14)`.

Next, all variables and complex operations on variables (e.g., pointer manipulations, structure operations, or unrecognized complex expressions) are normalized into **macro variables**. For example, an expression like `s->method->version == 0x303` is normalized into `x1 == 0x303`. This normalization ensures that the code slices are abstracted into a form suitable for semantic analysis.

2) SMT-Based Constraint Solving: Using the normalized code slices, we extract logical equations and expressions, which are then processed using the Z3 solver to check for semantic equivalence. To avoid interference from trivial or repetitive equations (e.g., `x1 == null`, `x1 == 0`), we extract the formulas that do not exist in the pre-patch version of the function as the unique features of the patched version. The same operation is also applied to the unique features of the pre-patch version. Only unique and meaningful equations are used for verification. If the Z3 solver finds semantically equivalent equations between the two slices, the verification is marked as successful.

3) LLM-Based Semantic Reasoning: For cases where SMT-based constraint solving cannot directly verify equivalence (e.g., lack of unique logical structures), we employ a Large Language Model (LLM) for semantic reasoning. The LLM analyzes the two matching results (pseudo code vs. vulnerable source code slice and pseudo code vs. patched source code slice) to infer semantic equivalence. The LLM is tasked with identifying whether the pseudo code slice represents the vulnerable functionality or the patched functionality. We constructed Prompt as shown in Table II. This reasoning step leverages the LLM's ability to understand high-level semantics and abstract relationships, providing a fallback mechanism for cases that are too complex for SMT solvers alone.

TABLE III
DATASET

Project	CVE	$func_v$	Version	Binary	Testcase
OpenSSL	34	100	20	400	2000
Freetype	7	18	5	100	360
Tcpdump	25	78	2	40	1560
Libxml2	7	28	4	80	560
All	73	224	31	620	4480

IV. IMPLEMENTATION

Lares is implemented in approximately 3,000 lines of Python code, integrating several tools for its functionality.

For binary analysis, we use IDA Pro [22] to decompile binaries and extract function-level pseudocode via custom scripts. IDA Pro was selected for its robust decompilation and scripting capabilities. For source code, Tree-Sitter [23] parses code into functions with AST-based segmentation, while Joern [21] performs advanced static analysis, including data flow and control flow, using its code property graph model.

Semantic equivalence is determined with the Z3 SMT solver [24], which verifies logical equivalences by normalizing code slices into macro-variables. For cases where formal methods are insufficient, Lares uses the Claude-3.5-Sonet [25] language model for semantic reasoning. Due to cost considerations, larger models like GPT-o1 were avoided, with vulnerability verification costing approximately $0.20 per case. We use the default values for LLM parameters (temperature=1.0).

V. EVALUATION

We evaluate the effectiveness of Lares by answering the following research questions.

RQ1: How does Lares perform in the cross-optimization patch presence testing task compared to existing methods?

RQ2: How does Lares perform in the cross-architecture patch presence testing task?

RQ3: How does every component in Lares affect the overall performance?

RQ4: How efficient is Lares?

a) Experiment setup.: The experiments are conducted on Ubuntu 22.04, powered by an Intel Xeon CPU with 128 cores at 3.0GHz and hyperthreading capabilities.

b) Evaluation metrics.: We follow the evaluation criteria established in existing works [9], [14], [15], and use precision, recall and F1 score to evaluate the performance of Lares and baselines. Specifically, TP, FP, and FN refer to the number of patched functions truly classified as patched, vulnerability functions erroneously classified as patched, and patched functions erroneously classified as vulnerability, respectively.

TABLE IV
PERFORMANCE OF PATCH PRESENCE TESTING WITH DIFFERENT OPTIMIZATION OPTIONS ON X86.

Model	O0			O1			O2			O3			Average			$Average_{new\_vul}$		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
BinXray	1.0	0.49	0.66	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Robin	0.64	0.75	0.69	0.63	0.70	0.67	0.62	0.69	0.66	0.63	0.73	0.68	0.63	0.72	0.67	0.69	0.73	0.70
PS3	0.76	0.93	0.83	0.56	0.70	0.62	0.56	0.74	0.64	0.56	0.74	0.64	0.61	0.78	0.68	0.62	0.79	0.69
Lares	0.69	0.91	0.79	0.71	0.77	0.74	0.72	0.79	0.75	0.78	0.83	0.81	**0.72**	**0.83**	**0.77**	0.76	0.81	0.79

TABLE V
PATCH PRESENCE TESTING ACCURACY OF DIFFERENT COMPILATION ENVIROMENT.

Enviroment	O0			O1			O2			O3			Average			$Average_{new\_vul}$		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
x86-gcc	0.66	0.85	0.75	0.71	0.79	0.74	0.68	0.77	0.72	0.73	0.79	0.76	0.69	0.80	0.74	0.74	0.79	0.76
x64-gcc	0.65	0.85	0.74	0.69	0.79	0.73	0.66	0.79	0.72	0.73	0.81	0.76	0.68	0.81	0.74	0.74	0.80	0.77
arm-gcc	0.62	0.80	0.70	0.64	0.75	0.69	0.69	0.75	0.72	0.70	0.76	0.73	0.66	0.76	0.71	0.68	0.74	0.71
x86-clang	0.69	0.91	0.79	0.71	0.77	0.74	0.72	0.79	0.75	0.78	0.83	0.81	0.72	0.83	0.77	0.76	0.81	0.79
x64-clang	0.68	0.91	0.78	0.70	0.77	0.73	0.71	0.76	0.73	0.73	0.79	0.76	0.71	0.81	0.76	0.74	0.81	0.77

c) Dataset: We select four real-world well-known projects from various application aspects for evaluation, which is the same as the existing works [14], [15]. These four projects involved protocol encryption, packet, XML analyzer, and font rendering. We compile the vulnerable and patched versions of OSS for each vulnerability collected using compiler gcc v9.4.0 and clang v6 with different optimization levels (O0 to O3) and different architectures (x86, x64, and ARM), respectively. Table III shows the statistical information for these projects. In total, we have obtained 73 CVEs with 224 distinct vulnerable and patched functions, denoted as the source functions. Finally, we conduct 20 * (112 + 112) = 4480 test cases.

In addition, we expanded the dataset to include 20 vulnerabilities disclosed after October 2024, totaling 400 cases, to ensure that they were not included in the training set of the model (claude-3.5-sonnet-20241022) to avoid data contamination.

d) Baselines: We select three baselines in our experiments.

- **BinXray [11]**: A classic syntactic-based approach for patch presence testing, widely adopted in various works.
- **Robin [9]**: A state-of-the-art semantic-based patch presence testing approach that leverages symbolic execution to extract patch features.
- **PS3 [14]**: A state-of-the-art patch presence testing approach specifically designed to address challenges posed by compiler optimization.

Additionally, REACT [15] needs to be compiled with clang to generate Intermediate Representations (IR). This process requires an additional compilation process after the normal compilation, which is more difficult to use than other methods. Since we aim to implement a compile-free method, REACT is orthogonal to our scope of consideration.

A. Cross-optimization Task (RQ1)

To evaluate Lares' performance in the cross-optimization (O0, O1, O2, and O3) patch presence testing task, we compare it against three state-of-the-art methods. The results are summarized in Table IV.

BinXray [11] exhibits high precision under O0 (1.0). H owever, it fails entirely under higher optimization levels (O1, O2, O3). This is because BinXray [11] heavily relies on syntactic features. Robin [9] demonstrates relative stability across different optimization levels, maintaining an F1 score in the range of 0.66 to 0.69. However, its overall accuracy and recall remain low compared to Lares. This is primarily because Robin [9] relies on symbolic execution for feature extraction, which fails for certain functions, and some extracted features are insufficient to distinguish between patched and unpatched code. PS3 [14] shows decent performance under O0, with an F1 score of 0.83, but its performance degrades significantly under higher optimization levels (F1 score drops to 0.62, 0.64, and 0.64 for O1, O2, and O3, respectively). This degradation is due to the presence of too many patch-irrelevant features in its analysis, which introduces noise and leads to a high rate of missed patch detections.

Lares outperforms all baseline methods, showing robust performance across all optimization levels with an average F1 score of 0.77. Unlike BinXray [11], Lares effectively handles cross-optimization scenarios by leveraging semantic features, which are less sensitive to syntactic changes introduced by optimization. Compared to PS3 [14], Lares reduces the influence of patch-irrelevant features through enhanced data flow, control flow, and macro analysis, improving recall even under high optimization levels. Finally, Lares surpasses Robin [9] by combining deterministic SMT-based verification with semantic reasoning, which enables it to handle complex cases where symbolic execution fails or extracted features are ambiguous.

Specifically, we also evaluated 400 new cases, denoted as $Average_{new\_vul}$. These new vulnerabilities are consistent with

the findings from conventional datasets, and Lares significantly improves over existing methods. This further demonstrates that Lares's improvement is not due to data contamination. This is consistent with our intuition, as Lares leverages the code analysis capabilities of LLM rather than its inherent understanding of vulnerabilities in the training set, and is therefore not affected by data contamination.

Answering RQ1: Lares can handle diverse optimization levels and significantly improve the precision and recall of existing methods.

B. Cross-architecture Task (RQ2)

To evaluate the performance of Lares in the cross-architecture and cross-compiler patch presence testing task, we tested it on three architectures (x86, x64, ARM) and two compilers (GCC and Clang) with different optimization levels. Table V summarizes the precision, recall, and F1 score of Lares under these configurations.

Existing methods such as BinXray [11], PS3 [14], and Robin [9] rely heavily on architecture-specific or compiler-specific features, limiting their ability to handle cross-architecture or cross-compiler scenarios. They are strongly related to instruction tokens, while different architectures have different instruction sets. Therefore, we only present the results of Lares. Unlike existing methods, Lares is explicitly designed to generalize across architectures and compilers.

Lares demonstrates consistent performance across the five settings. For x86 and x64 compiled with GCC, it achieves an average F1 score of 0.74, while for ARM-GCC, the score is also 0.71. Similarly, under Clang, the F1 scores range from 0.75 to 0.77. The slight variations across architectures are primarily due to differences in instruction sets and binary representations, but these have minimal impact on Lares' overall performance. Clang-based binaries show a slight improvement in F1 scores, possibly due to better preservation of semantic features during compilation. Similarly, IDA-Pro's decompilation for x86 architecture is more mature than ARM, so the result is slightly higher.

Answering RQ2: Lares outperforms existing methods by effectively handling the cross-architecture and cross-compiler patch presence testing task.

C. Ablation Evaluation (RQ3)

We evaluated Lares with different LLMs and temperature parameters. We have repeated every case 5 times when the temperature is not zero and observed no significant changes in the results. As shown in Table VI, Claude-3.5-Sonet achieved the highest F1 score, followed by GPT-4o. The open-source model DeepSeek-Coder exhibited the highest both Recall but the lowest Precision. Claude-3.5 and GPT-4o significantly outperformed existing methods, while other models showed comparable performance. More advanced models, such as o1, were excluded due to cost constraints.

TABLE VI
PERFORMANCE OF DIFFERENT LLMS ACROSS SETTINGS. DARKER GREEN INDICATES BETTER PERFORMANCE

		Temperature			
	Metrics, Model	**0.0**	**0.5**	**0.7**	**1.0**
Precision	GPT-3.5-Turbo	0.514	0.526	0.543	0.525
	DeepSeek-Coder	0.511	0.510	0.515	0.536
	GPT-4o	0.586	0.598	0.618	0.647
	Claude-3.5-Sonnet	0.629	0.665	0.706	0.723
Recall	GPT-3.5-Turbo	0.814	0.826	0.829	0.792
	DeepSeek-Coder	0.983	0.934	0.951	0.968
	GPT-4o	0.824	0.923	0.885	0.872
	Claude-3.5-Sonnet	0.673	0.802	0.769	0.831
F1 Score	GPT-3.5-Turbo	0.631	0.643	0.656	0.631
	DeepSeek-Coder	0.673	0.660	0.668	0.690
	GPT-4o	0.685	0.726	0.728	0.743
	Claude-3.5-Sonnet	0.651	0.727	0.736	0.773

TABLE VII
ABLATION STUDY

Model	Precision	Recall	F1
$Lares_{-pe}$	0.651	0.720	0.683
$Lares_{-pl}$	0.488	0.600	0.538
$Lares_{-z3}$	0.673	0.780	0.714
$Lares_{-pv}$	0.815	0.221	0.348
$Lares$	0.725	0.825	0.772

To evaluate the contribution of each module in Lares, we conducted an ablation study by progressively removing individual components. The results are summarized in Table VII, where the precision, recall, and F1 score of different variants of Lares are reported. The settings are same as Table IV.

Removing the patch enhancement module, $Lares_{-pe}$, degenerates into a common LLM-based function similarity matching method, which results in a precision of 0.651, recall of 0.720, and an F1 score of 0.683. This decline is due to the failure of the LLM to distinguish subtle differences in certain patches without the enriched semantic information provided by this module. When the patch localization module ($Lares_{-pl}$) is removed, and the LLM is directly tasked with detecting the presence of patches from pseudo code, the performance drops significantly, with an F1 score of 0.538. The absence of patch localization results in the LLM needing to process raw pseudo code with many missing variable names and structural ambiguities, which makes it difficult to reliably determine patch presence. This highlights the importance of the patch localization module in narrowing down the pseudo code to relevant slices, enabling the LLM to focus on smaller, semantically meaningful code regions.

Removing the Z3 solver ($Lares_{-z3}$) leads to a slight performance drop, with an F1 score of 0.714. Z3 is important and can improve F1-score from 71.4% to 87.3%. However, the F1-score for Lares is 77.3%(not 87.3%) because Z3 only handles

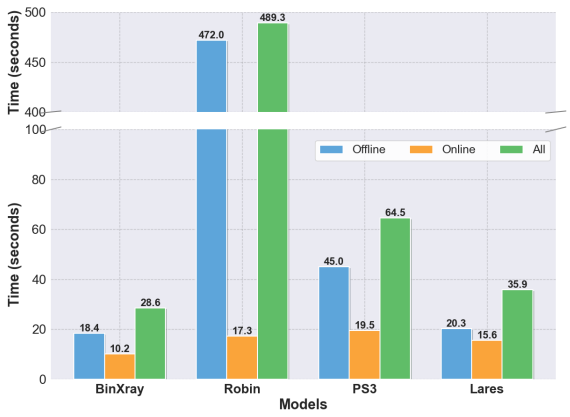

Fig. 5. Timecost Evaluation (seconds).

21% testcases; the others, without equations for Z3 to prove semantic equivalence, are handled by LLM. Therefore, Z3 is necessary because when it works, the results will be highly confident. LLM is complementary to advancing scalability. In addition, the accuracy of LLM cannot be ignored. Although the LLM is less accurate than Z3, LLM-only performance still surpasses PS3 [14] and Robin [9].

Removing the LLM-based patch verification ($Lares_{-pv}$) and relying solely on Z3 for verification results in significant performance changes. While the precision of $Lares_{-pv}$ is high (0.815), the recall drops substantially to 0.221, as only 21% of test cases can be successfully verified by the Z3 solver. This limitation stems from Z3's dependency on patches that can be fully modeled using logical constraints, whereas many real-world patches lack unique semantic equations suitable for Z3's processing. These results highlight that, although Z3 achieves high precision, its applicability is limited, requiring LLM to deal with more cases.

Answering RQ3: Leveraging LLM, Lares significantly surpasses existing methods. Each component contributes critically to its overall performance.

D. Efficiency (RQ4)

To evaluate the efficiency of Lares, we compared its runtime with existing approaches. The runtime is divided into two phases: *OffLine*, which includes offline preprocessing tasks such as reverse engineering and feature extraction for patches, and *OnLine*, which refers to the patch presence testing time for each input binary. The results are summarized in Figure 5.

BinXray [11] has the shortest runtime, with a total time of 28.6 seconds. This is because it uses only simple syntactic features. Robin [9] is the most time-consuming among all methods, with a total runtime of 489.31 seconds. The primary bottleneck is the *OffLine* phase, which takes 472 seconds due to the use of symbolic execution for feature extraction. PS3 [14] improves efficiency compared to Robin [9] by simplifying symbolic execution into symbolic simulation, significantly reducing the time required in the *OffLine* phase to 45 seconds.

Lares achieves a total runtime of 35.9 seconds, which is significantly faster than Robin [9] and PS3 [14] and close to the lightweight BinXray [11]. Its *OffLine* phase takes only 20.3 seconds, as it avoids the computational overhead of symbolic execution by leveraging semantic analysis based on a large language model (LLM). The *OnLine* phase (15.6 seconds) is also reasonably fast, enabling efficient patch detection without compromising accuracy.

The runtime comparison in Figure 5 does not include the time required for compiling binaries. BinXray [11], Robin [9] and PS3 [14] require a compilation process to construct patch features, which adds significant overhead in real-world use.

Answering RQ4: Lares outperforms existing methods and particularly suitable for scenarios where speed and adaptability are crucial.

E. Failure Cases Analysis

We investigate various failure cases in this experiment and identify several potential causes behind these inaccuracies:

- **Repeated patch code**: The statements added by patches that are also in the vulnerable function, but in different locations to interfere with judgment. For example, the if statement "if (s $\rightarrow$ `session` $\rightarrow$ `sess_cert` == `NULL`)" in CVE-2014-3510 and the code in the body also appear elsewhere in the vulnerable function. This is a relatively common assertion. Therefore, Lares also recognizes this statement in the vulnerable function and mistakenly identifies the vulnerable function as patched. This situation is rare in our dataset. In the future, researchers can optimize the Patch Enhancement module to avoid this failure by adding more context.
- **Stealthy patches**: Stealthy patches often introduce only minor changes. Variations from different compiler environments further obscure these differences, making accurate judgment difficult. In fact, these cases are also difficult for humans. Specifically, the LLM shows high precision for add/delete-type patches but occasionally fails on edit-type patches by missing subtle differences or misidentifying changes.
- **The hallucinations of LLM**: Hallucination is one of the recognized problems in LLM. Hallucinations can degrade performance by causing missed patches or incorrect judgments, especially in logical comparisons and complex vulnerabilities. For instance, in CVE-2013-6449 the condition changed from ">=" to "==" post-patch, yet the LLM sometimes hallucinated a "==" and falsely deemed the vulnerability patched. Future work will mitigate hallucinations via iterative verification and building more complex multi-agent workflows.

In summary, Lares demonstrated that the agent built by combining LLM and Z3 has considerable potential in the patch existence testing task. Lares cannot achieve perfect judgment

in some special cases due to the presence of hallucinations. However, we can try to address these issues in the future by improving the agent or designing workflows based on vulnerability types.

VI. Discussion

Patch existence testing, which involves detecting fine-grained patch semantics, remains a challenging task with significant room for improvement in existing methods, including Lares. Lares offers interpretable outputs, such as equations generated by Z3 or reasoning provided by LLMs, which can serve as valuable references for humans, even in cases of uncertain results.

In this work, we employed a zero-shot commercial LLM for patch presence testing and achieved promising results. However, this approach incurs a certain API usage cost. In the future, by fine-tuning open-source LLMs on patch-related data, it is possible to further enhance detection performance while reducing the cost of deployment. Besides, Sometimes Lares will return a result that is not in JSON format because of the randomness of LLM, and we need to ask again.

Our dataset includes a diverse set of vulnerable and patched functions, comprising different types of projects. This dataset is consistent with previous work [14] but goes beyond by generating significantly more test cases, with a total of 4480 samples. The larger dataset ensures a more thorough and reliable evaluation of Lares' performance, covering a wide range of real-world scenarios and challenges.

VII. Related Work

A. Vulnerability Detection

Binary code similarity detection plays a crucial [26], [27] role in applications like malware analysis [28], vulnerability detection [29], [30], and software reuse identification [6], [19], [31], [32]. This work focuses on detecting 1-day vulnerabilities caused by binary code reuse [4], [5], [33], [34]. The main challenge is to overcome the differences caused by different compilation environments and identify semantic equivalence.

Recent works leverage Natural Language Processing (NLP) techniques and structural analysis for binary similarity detection. NLP-based methods, such as SAFE [35], Trex [36], and CEBin [37], treat code as natural language and embed semantics for matching. Structural approaches like Gemini [3], [38], ASTeria [39], and HermesSim [40] exploit control flow graphs (CFG), abstract syntax trees (AST), or custom semantic graphs for similarity detection.

Large language models (LLMs) have recently emerged as a new paradigm in this field. Methods like CLAP [41] and SCALE [42] use LLMs to enhance similarity detection.

Despite these advances, current methods struggle to determine if detected vulnerabilities have been patched. To address this, our work targets patch presence testing, which complements vulnerability detection by verifying whether a vulnerability has been mitigated, offering a more comprehensive assessment of software security.

B. Vulnerability Verification

To address false positives in vulnerability detection, verification methods are employed, including dynamic and static approaches.

Dynamic methods, such as Directed Grey-box Fuzzing (DGF) [43]–[47], generate proofs of concept (PoC) by fuzzing target binaries [48]. AFLGo [49] is a classic DGF approach. Subsequent methods like Hawkeye [50], WindRanger [51], and TransferFuzz [52] build upon AFLGo, enabling direct vulnerability triggering. However, these methods are time-consuming and limited to a small subset of memory vulnerabilities.

To overcome these limitations, static methods like patch existence testing have been proposed. These approaches analyze patch modification statements to detect whether a function remains vulnerable or has been patched. Some researchers [53], [54] have analyzed the evolution of vulnerability patches and classified various security patches. Patch existence testing methods [55] for Java, Android, or C/C++ source code are relatively mature. However, patch existence testing for C/C++ binary targets brings unique challenges due to different compilation environments. BinXray [11] uses lightweight syntactic features. Robin [9] and PS3 [14] leverage symbolic execution and simulation for higher accuracy but face scalability challenges. Specialized kernel-focused methods, such as Fiber [10] and PDiff [13], have also been developed.

Existing approaches often trade off accuracy for usability or vice versa. We propose *Lares*, a lightweight method that achieves both high usability and accuracy while uniquely supporting cross-optimization, cross-architecture, and cross-compiler scenarios.

VIII. Conclusion

It is crucial to determine whether the target function in binary has been patched. In this paper, we presented Lares, a novel method for patch presence testing. Unlike traditional methods that rely on compilation process to extract patch features, Lares directly leverages source code functions and patch information, providing a lightweight and scalable solution for cross-architecture detection. By employing LLM-driven semantic analysis, Lares accurately identifies patch code slices, significantly improving precision and recall. We compare Lares with existing methods in detail and discuss the contribution of each component of Lares. Our evaluation demonstrates that Lares not only achieves high scalability but also effectively handles diverse architectures and compilation environments.

IX. Acknowledge

This work was supported by Key R&D Program of Shandong Province, China(No. 2024CXGC010114), National Natural Science Foundation of China under Grant(No. 62372268), Shandong Provincial Natural Science Foundation, China (No. ZR2022LZH013, No. ZR2021LZH007). Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, et al. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 442–454, 2022.

[2] Synopsys. Synopsys 2024 open source security and risk analysis report., 2024. https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html.

[3] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[4] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2022.

[5] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *NDSS*, 2023.

[6] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. Libdb: an effective and efficient framework for detecting third-party libraries in binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 423–434, 2022.

[7] Siyuan Li, Yongpan Wang, Chaopeng Dong, Shouguo Yang, Hong Li, Hao Sun, Zhe Lang, Zuxin Chen, Weijie Wang, Hongsong Zhu, et al. Libam: An area matching framework for detecting third-party libraries in binaries. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–35, 2023.

[8] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. Binaryai: Binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[9] Shouguo Yang, Zhengzi Xu, Yang Xiao, Zhe Lang, Wei Tang, Yang Liu, Zhiqiang Shi, Hong Li, and Limin Sun. Towards practical binary code similarity detection: Vulnerability verification via patch semantic analysis. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[10] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, 2018.

[11] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387, 2020.

[12] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, Zhaohui Zhou, Haijun Wang, and Ting Liu. Patchdiscovery: Patch presence test for identifying binary vulnerabilities based on key basic blocks. *IEEE Transactions on Software Engineering*, 49(12):5279–5294, 2023.

[13] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1149–1163, 2020.

[14] Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. Ps3: Precise patch presence test based on semantic symbolic signature. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[15] Qi Zhan, Xing Hu, Xin Xia, and Shanping Li. React: Ir-level patch presence test for binary. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 381–392, 2024.

[16] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*, 2024.

[17] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/-CorpusID*, volume 267622140, 2024.

[18] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. B2sfinder: detecting open-source software reuse in cots software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1038–1049. IEEE, 2019.

[19] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. Interpretation-enabled software reuse detection based on a multi-level birthmark model. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 873–884. IEEE, 2021.

[20] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.

[21] Joern., 2014. https://joern.io/.

[22] Ida pro., 2023. https://hex-rays.com/IDA-pro/.

[23] tree-sitter., 2017. https://github.com/tree-sitter/tree-sitter.

[24] Z3 prover., 2023. https://github.com/Z3Prover/z3.

[25] Claude., 2023. https://claude.ai/.

[26] Saed Alrabaee, Mourad Debbabi, and Lingyu Wang. A survey of binary code fingerprinting approaches: Taxonomy, methodologies, and features. *ACM Computing Surveys (CSUR)*, 55(1):1–41, 2022.

[27] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Computing Surveys (CSUR)*, 54(3):1–38, 2021.

[28] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. {BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 253–270, 2017.

[29] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359, 2017.

[30] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[31] Siyuan Li, Chaopeng Dong, Yongpan Wang, Wenming Liu, Weijie Wang, Hong Li, Hongsong Zhu, and Limin Sun. Libdi: A direction identification framework for detecting complex reuse relationships in binaries. In *MILCOM 2023-2023 IEEE Military Communications Conference (MILCOM)*, pages 741–746. IEEE, 2023.

[32] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 104–115. IEEE, 2020.

[33] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR, 2021.

[34] Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. Asteria-pro: Enhancing deep-learning based binary code similarity detection by incorporating domain knowledge. *arXiv preprint arXiv:2301.00511*, 2023.

[35] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[36] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[37] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. Cebin: A cost-effective framework for large-scale binary code similarity detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 149–161, 2024.

[38] Chaopeng Dong, Siyuan Li, Shouguo Yang, Yang Xiao, Yongpan Wang, Hong Li, Zhi Li, and Limin Sun. Libvdiff: Library version difference guided oss version identification in binaries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based ast-encoding

for cross-platform binary code similarity detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 224–236. IEEE, 2021.

[40] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*, 2024.

[41] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. Clap: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 503–515, 2024.

[42] Xin-Cheng Wen, Cuiyun Gao, Shuzheng Gao, Yang Xiao, and Michael R Lyu. Scale: Constructing structured natural language comment trees for software vulnerability detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 235–247, 2024.

[43] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. {DAFL}: Directed grey-box fuzzing guided by data dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4931–4948, 2023.

[44] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*, pages 167–182, 2017.

[45] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[46] Changhua Luo, Wei Meng, and Penghui Li. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2693–2707, 2023.

[47] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[48] Seongkyeong Kwon, Seunghoon Woo, Gangmo Seong, and Heejo Lee. Octopocs: automatic verification of propagated vulnerable code using reformed proofs of concept. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 174–185. IEEE, 2021.

[49] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[50] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2095–2108, 2018.

[51] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2440–2451, 2022.

[52] Siyuan Li, Yuekang Li, Zuxin Chen, Chaopeng Dong, Yongpan Wang, Hong Li, Yongle Chen, and Hongsong Zhu. Transferfuzz: Fuzzing with historical trace for verifying propagated vulnerability code. *arXiv preprint arXiv:2411.18347*, 2024.

[53] Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. Patch correctness assessment: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–50, 2025.

[54] Ruyan Lin, Yulong Fu, Wei Yi, Jincheng Yang, Jin Cao, Zhiqiang Dong, Fei Xie, and Hui Li. Vulnerabilities and security patches detection in oss: a survey. *ACM Computing Surveys*, 57(1):1–37, 2024.

[55] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. Precise and efficient patch presence test for android applications against code obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 347–359, 2023.