What Do They Fix? LLM-Aided Categorization of Security Patches for Critical Memory Bugs

Xingyu Li*, Juefei Pu*, Yifan Wu*, Xiaochen Zou*, Shitong Zhu*, Qiushi Wu§, Zheng Zhang*, Joshua Hsu*, Yue Dong*, Zhiyun Qian*, Kangjie Lu†, Trent Jaeger*, Michael De Lucia‡, Srikanth V. Krishnamurthy*
* UC Riverside † University of Minnesota § IBM ‡ U.S. Army Research Laboratory
* {xli399,jpu007,fshal003,xzou017,szhu014,zzhan173,jhsu094,yued,trentj,zhiyun.qian,krish}@ucr.edu
† kjlu@umn.edu § Qiushi.Wu@ibm.com ‡ michael.j.delucia2.civ@army.mil

***Abstract*—Open-source software projects are foundational to modern software ecosystems, with the Linux kernel standing out as a critical exemplar due to its ubiquity and complexity. Although security patches are continuously integrated into the Linux mainline kernel, downstream maintainers often delay their adoption, creating windows of vulnerability. A key reason for this lag is the difficulty in identifying security-critical patches, particularly those addressing exploitable vulnerabilities such as out-of-bounds (OOB) accesses and use-after-free (UAF) bugs. This challenge is exacerbated by intentionally silent bug fixes, incomplete or missing CVE assignments, delays in CVE issuance, and recent changes to the CVE assignment criteria for the Linux kernel.**

Prior efforts such as GraphSPD, have proposed binary classifiers to distinguish security versus non-security patches. However, these approaches do not provide fine-grained categorization of vulnerability types, which is essential for prioritizing fixes for high-impact bugs like OOB and UAF. Our work aims to take such coarsely labeled security patches and classify them into fine-grained categories, i.e., OOB, UAF, or non-OOB-UAF types.

While fine-grained patch classification approaches exist, they exhibit limitations in both coverage and accuracy. In this work, we identify previously unexplored opportunities to significantly improve fine-grained patch classification. Specifically, by leveraging cues from commit titles/messages and diffs alongside appropriate code context, we develop DUALLM, a dual-method pipeline that integrates two approaches based on a Large Language Model (LLM) and a fine-tuned small language model. DUALLM achieves 87.4% accuracy and an F1-score of 0.875, significantly outperforming prior solutions. Notably, DUALLM successfully identified 111 of 5,140 recent Linux kernel patches as addressing OOB or UAF vulnerabilities, with 90 true positives confirmed by manual verification (many do not have clear indications in patch descriptions). Moreover, we constructed proof-of-concepts for two identified bugs (one UAF and one OOB), including one developed to conduct a previously unknown control-flow hijack as further evidence of the correctness of the classification.

I. INTRODUCTION

Open-source software projects have emerged as cornerstones of modern software development ecosystems. However, incorporating such projects leads to significant security risks. Memory safety vulnerabilities are particularly notable due to their prevalence and severity. However, *memory safety vulnerabilities vary significantly in severity*. Among them, out-of-bounds (OOB) access and use-after-free (UAF) vulnerabilities (including double-free and invalid-free) are especially perilous, as malicious actors consistently exploit them to achieve privilege escalation and compromise system integrity [19], [6], [3], [1]. In fact, our analysis of publicly available exploits [14] since 2020 showed that almost **90%** of them exploit OOB or UAF vulnerabilities.

Taking the Linux kernel as an example, it has been shown that even when patches addressing such vulnerabilities are promptly integrated into the upstream mainline kernel, downstream kernel maintainers (e.g., Ubuntu) may delay their adoption for weeks or even months [44], [78]. This extended lag creates a substantial window of vulnerability in which attackers may exploit bugs to attack unpatched downstreams. Moreover, studies have demonstrated the feasibility of automated exploit generation targeting OOB and UAF bugs [27], [71], [66], [73], further underscoring the urgent need for timely patch porting.

Prior work (e.g., [78]) has documented this issue extensively, which identifies a major cause of this delay: downstream maintainers often lack clear, timely, and fine-grained information about which patches are security-critical, i.e., OOB or UAF bugs [78], [85]. The volume of daily commits makes it infeasible to manually audit every patch. Furthermore, CVE-based indicators are insufficient: (1) many security patches are fixed silently without any indications (i.e., perhaps no one understood the security impact); (2) CVE assignments often lag behind the availability of patches by weeks or even months [41]; (3) CVEs are incomplete — many exploitable security vulnerabilities are not assigned CVE numbers [28], [85], [45], [84].

While several tools have been proposed to detect security-related patches (e.g., VulFixMiner [81], GraphSPD [62]), they generally offer only binary classification (security vs. non-security), which is insufficient for practical prioritization, as not all vulnerabilities are equally impactful and urgent to be ported. To enable downstream maintainers to triage and prioritize critical patches earlier, a more useful analysis involves a *fine-grained classification* that distinguishes between vulnerability types, particularly to tease out those with high

Network and Distributed System Security (NDSS) Symposium 2026
23–27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230328
www.ndss-symposium.org

exploitability such as UAF and OOB. Although several approaches have previously been proposed, all exhibit significant limitations in both coverage and accuracy.

Limitations of Prior Art. Specifically, state-of-the-art solutions SID [70], TreeVul [49] and CoLeFunDa [80] offer fine-grained patch classifications. SID [70] relies on human-defined, hard-coded patterns, but the complexity and variety of patterns in the real world make them difficult to define and accurately capture. Specifically, it supports only a single hard-coded patch pattern for each UAF and OOB vulnerability type, resulting in low coverage, and fails to identify 57% of the relevant patches. TreeVul [49] and CoLeFunda [80] eliminate the need for hard-coded patterns by feeding the code diffs (e.g., added and removed lines of code) to a machine-learning model. Unfortunately, TreeVul does not use any code context beyond the diff itself, limiting the model's ability to appropriately learn patch patterns. While ColeFunda does use code context, its use of the standard slicing technique can introduce bloated code blocks and noise. Furthermore, both approaches suffer from a common limitation: they do not leverage commit messages, missing valuable semantic cues in natural language that can aid classification.[1] Our evaluations show that (see §VI), TreeVul achieves an accuracy of 65.6 % and an F1-score of 0.653, in identifying patches that fix UAF and OOB bugs, indicating ample room for improvement. These limitations motivate the need for a more effective solution that advances the state of the art in vulnerability type classification.

Our solution. To tackle the challenge of handling diverse patch patterns, we propose a novel machine-learning-based solution guided by two key insights: First, beyond code diffs in a patch, the commit titles and messages often provide valuable insights into the nature of the bug being fixed (e.g., directly mentioning a "use-after-free" bug being fixed). The advancements in LLMs make them very suitable for extracting such indicators from commit descriptions. Different from CoLeFunDa [80] and TreeVul [49], our approach leverages Large Language Models (LLMs) to effectively extract and utilize these valuable indicators from commit descriptions. While an LLM-only pipeline already outperforms recent approaches such as SID and TreeVul, its accuracy still falls short of our expectations. Second, critical clues are often found in the code context that extends beyond the modified lines in a patch that can help identify the type of bug. We develop a custom slicing method that concisely captures the impact of the patch to drive a fine-tuned small language model (since our slices are custom and specialized). Finally, we develop a dual-method pipeline, namely DUALLM, that runs end-to-end automatically to classify patches into UAF, OOB, and other non-UAF-OOB patches, leveraging both types of information in an informed way. We note that our approach differs from prior approaches relying on a single machine learning model, including CoLeFunDa [80] and TreeVul [49]. In contrast, our method adopts a hybrid design that strategically integrates a large language model (LLM) with a small fine-tuned language model, effectively harnessing the strengths of both to achieve significantly higher classification performance.

We evaluated DUALLM primarily on Linux kernel patches, achieving an 87.4% accuracy and a 0.875 F1-score on quality-controlled CVE patches, outperforming SID and TreeVul by **23.6%** and **21.2%** in accuracy, and 0.329 and 0.216 in F1-score. Most importantly, out of 5,140 recent patches, DUALLM identified 111 as fixing OOB or UAF bugs, with manual verification confirming **90** of these identifications as true positives. This highlights the significant coverage of DUALLM in teasing out critical security patches. Notably, we construct proof-of-concepts for two such bugs (one UAF and one OOB), as further evidence of the correctness of the classification. We even successfully exploited one such bug to realize a control-flow hijack attack that was not publicly known.

Scope and assumptions. To show the generality of DUALLM, we also tested our method against other types of bugs (e.g., NULL pointer dereference, memory leak, and use-before-initialization) and additional open-source projects, yielding similarly robust results. Consistent with TreeVul and CoLeFunDa, we assume the patches fed to our fine-grained classifier are already security patches (e.g., CVE patches). If they are not already labeled as such, we envision a full pipeline to run existing coarse-grained classifiers that differentiate security and non-security patches, e.g., VulFixMiner [81] and GraphSPD [62]. This is different from SID which does not require the input to be security patches.

Contributions. A summary of our contributions is as follows:

- We develop DUALLM, a system that strategically leverages the strengths of an LLM and a specialized small language model to effectively classify security patches addressing critical memory corruption bugs, specifically targeting OOB and UAF vulnerabilities.
- Our solution is driven by missed opportunities and limitations in state-of-the-art solutions, as well as recent advances in large language models, including leveraging an LLM to enhance vulnerability context extraction during slicing.
- We show that DUALLM achieves an 87.4% accuracy and a 0.875 F1-score on quality-controlled CVE patches, markedly outperforming state-of-the-art methods [70], [49], and identifies 90 OOB and UAF patches from 5,140 recent patches. From these, we further develop two PoCs and one exploit to achieve control flow hijacking.
- We will open source the code, data, and model produced as part of the research to facilitate the reproduction of the results and further research.

II. BACKGROUND AND RELATED WORK

A. Memory Corruption Vulnerabilities

Memory corruption vulnerabilities remain among the most critical security threats in modern software systems, as they can enable attackers to achieve arbitrary code execution and

[1] CoLeFunDa has three downstream tasks, what we refer for it in the paper is its CWE classification task. Also, CoLeFunDa is not open-sourced.

escalate privileges. These vulnerabilities can arise at any stage of a memory object's lifecycle—from allocation and initialization to usage and eventual deallocation—creating multiple opportunities for exploitation.

Among memory corruption vulnerabilities, out-of-bounds access and use-after-free are particularly critical due to their high exploit potential for arbitrary code execution and privilege escalation [71], [27], [66], [73]. Our analysis of 69 publicly available exploits [14] targeting Linux/Android systems since 2020 reveals that 38 were UAF vulnerabilities and 24 were OOB vulnerabilities, with only 7 being non-UAF-OOB bugs.

Security vulnerabilities are typically documented using the Common Vulnerabilities and Exposures (CVEs) [4] system. Each CVE is typically categorized using the Common Weakness Enumeration (CWE), a standardized taxonomy that classifies different types of software weaknesses and vulnerabilities. For example, CWE-787 represents the category of out-of-bounds write vulnerabilities where programs write beyond the bounds of allocated memory regions, while CWE-416 represents use-after-free vulnerabilities where programs attempt to use memory after it has been freed.

B. Mining Patches

Several studies [38], [69], [58] have used machine learning to assess whether a commit in the Linux mainline is a patch or a functional change, useful for patch porting decisions for stable/LTS branches. Once bug-fixing patches have been identified, the next challenge is to determine if such patches are security related. Many studies [82], [83], [86], [63], [65], [34], [54], [37], [81], [47], [57] have used machine learning to analyze patches to determine whether they are security related. For example, VulFixMiner [81] employs a transformer-based model and GraphSPD [62] leverages a graph neural network to perform coarse-grained patch classification (e.g., security vs. non-security). However, neither approach identifies the specific bug type, which is often critical for effective patch triaging [70], [80].

Some works (e.g., VFCFinder [30]) focus on mapping vulnerabilities to their corresponding patch commits. However, this assumes the availability of vulnerability disclosures or CVEs, which is not always the case—especially for silent fixes. Our goal is fundamentally different: we aim to proactively classify all patches into fine-grained vulnerability types, even in the absence of prior CVE assignment.

Generally, patches addressing different categories of vulnerabilities exhibit distinct characteristics in their code modifications. Patches fixing memory corruption vulnerabilities typically involve memory management operations, such as adding bounds checks before memory accesses, nullifying pointers, or ensuring proper object lifetime management through careful allocation and deallocation.

Based on such observations, SID [70] is among the first to identify such patch patterns for OOB, UAF, and two other bug types, using pre-defined rules. The key idea is to leverage under-constrained symbolic execution to analyze the security impact before and after the patch. While the solution is highly precise—rarely producing false positives—its performance is greatly limited by its reliance on a highly restricted set of hard-coded patterns (e.g., only one for each bug type). Even from among supported cases, it misses 53% of relevant patches due to the rigidity of its matching logic. Worse, patches that do not conform to any predefined pattern go undetected, leading to very high numbers of false negatives.

Besides SID, there are several machine-learning-based solutions proposed to classify patches in recent years [49], [80]. They both attempt to classify patches into CWE labels, which include OOB and UAF bugs. TreeVul [49] leverages a pre-trained CodeBert [32] model and further fine-tunes the same for this task. Its input is the code diff in the patch, i.e., the added lines and removed lines, without any code context. In comparison, CoLeFunDa [80] leverages contrastive learning and pre-trains a Bert model from scratch and fine-tunes it. Its input consists of program slices from the patched function. Neither solution leverages the patch descriptions, i.e., commit titles and messages, for classification. Both solutions also have only modest success. TreeVul's performance on Linux kernel patches is modest, with only a 66.98% accuracy and an F1-score of 0.670. Similarly, CoLeFunDa achieved a precision of 0.52 an F1-score of 0.50, as reported in their paper.

Our work addresses this gap by introducing DUALLM —a novel pipeline that combines large and small language models to achieve fine-grained vulnerability classification with significantly improved accuracy, *even for patches without assigned CVEs or clear descriptions.*

C. Machine Learning for Code Analysis

Transformer-based architectures have been successful in NLP [60], [52], [53], [25], [29], [40] and code understanding [32], [36], [21]. Recent work has applied these models to a range of security applications. For example, VulExplainer [33] attempts to classify a given vulnerable function into associated CWEs (vulnerability types). Sun *et al.* [56] proposed an encoder-decoder framework that aims to detect and explain security patches (not explicitly focusing on classification). Yu *et al.* [77] proposed hybrid models to perform binary code similarity detection.

The recent popularity of large language models [25], [26], [10], [48], [11], [15], [16], [17] has motivated researchers to investigate the use of LLMs for code related tasks. They include, but are not limited to, patch generation [72], [50], [22], [23], fuzz testing [39], [76], [75], vulnerability detection [59], [61], [55], bug reproduction [31], and assisting static analysis [42], [43]. While these studies demonstrate LLMs' potential across various code-related tasks, the specific challenge of identifying fine-grained vulnerability types such as OOB and UAF bugs from patches, remains unexplored.

III. MOTIVATION AND OVERVIEW

A. Motivation

Our approach is motivated by two crucial observations from preliminary studies of security patch classification. Before

ksmbd: fix use-after-free bug in smb2_tree_disconect
smb2_tree_disconnect() freed the struct ksmbd_tree_connect,but it left the dangling pointer. It can be accessed again under compound requests.

Fig. 1: Commit title and message of a Linux kernel patch with an explicit indication of the type of bug fixed

ipv6: raw: Deduct extension header length in rawv6_push_pending_frames
The total cork length created by ip6_append_data includes extension headers, so we must exclude them when comparing them against the IPV6_CHECKSUM offset which does not include extension headers.

Fig. 2: Commit title and message of a second Linux kernel patch with an implicit hint of the type of bug fixed

detailing our complete methodology, we first discuss these critical observations, as they form the foundation of our technical approach and showcase the novelty of our contribution. These observations not only reveal significant limitations in existing approaches but also guide our novel solution design.

Observation 1: Untapped potential exists in commit descriptions. While it is known that the natural language descriptions in patches can reveal the nature of issues being fixed, they have been leveraged thus far for only coarse-grained patch classification (in the pre-LLM era), i.e., into security vs. non-security patches, like PatchRNN [65] and other works [83], [86], [82], [54]. State-of-the-art methods, like CoLeFunDa [80],TreeVul [49] and SID [70], for fine-grained patch classification, e.g., OOB vs. UAF bugs, however, *completely overlook such valuable indicators.*

We observe that patch commit descriptions can sometimes directly reveal the exact bug type with explicit indicators (e.g., "use-after-free") as shown in Figure 1. In other cases, the bug type is not directly mentioned, but can be inferred from the description (from a human perspective). Figure 2 illustrates such an example where a miscalculation of a length variable occurs. Although the bug type is not explicitly specified, the length variable adjustment is indicative of a potential out-of-bounds access vulnerability. Surprisingly, our analysis of SID's dataset reveals the prevalence of such indicators: among the 227 security bugs that SID ultimately identified, our experiment shows that a remarkable **90.75%** contain direct or indirect indicators of the vulnerability type in their commit titles or messages, suggesting the utility of patch descriptions.

Observation 2: Code diff patterns cannot effectively identify bug types in prior research. In the cases where no clear indicators are available in the patch description, e.g., Figure 3 and Figure 8, code diff becomes a necessary signal for patch classification. For example, an out-of-bounds (OOB) vulnerability is often patched by adding a bounds check (see Figure 4a), or by recalculating the size of critical memory areas (see Figure 4b).

Don't feed anything but regular iovec's to blk_rq_map_user_iov
In theory we could map other things, but there's a reason that function is called "user_iov". Using anything else (like splice can do) just confuses it.

Fig. 3: Commit title and message of one Linux kernel patch without hints of the type of bug fixed

However, some patches do not follow common patch patterns and thus require additional code context beyond immediate code changes (an example is shown in Figure 8 discussed in detail later). This reveals a fundamental challenge in patch classification: *the true nature of a vulnerability fixed by a patch — its root cause, impact, and security implications — is often intertwined in intricate code relationships that extend far beyond the immediate patch vicinity.* Consequently, correctly classifying such patches to their bugs requires precisely extracting the full bug-logic-relevant code context.

Previous approaches [70], [49], [80] unfortunately have not developed an informed solution. SID [70] relies on symbolic rules (which are analyzed using symbolic execution) and support only a single patch pattern for each UAF and OOB vulnerability type. TreeVul [49] uses only the added lines and removed lines as the input to a machine learning model, without any additional code context — insufficient to identify bug types for some cases. Colefunda [80] does feed additional lines of code to a machine learning model via intra-procedural slicing. However, the standard slicing technique will capture irrelevant code context due to control dependence and lack of vulnerability context that may span function boundaries. We discuss these limitations in more detail in §IV-B.

Summary: The above observations reveal new opportunities to improve the classification of security patches, which directly motivate the design of our solution outlined in the next section.

B. Solution Overview

In this section, we present an overview of the key components of DUALLM and highlight three core design strategies.

Design strategy 1: Classifying patches with an LLM. We argue that LLMs offer a superior solution for classifying patches compared to prior approaches. Recent advancements have demonstrated the exceptional ability of LLMs to handle both natural language and code-related tasks [79], [48], [51], [72]. This makes them particularly well-suited for (1) analyzing patch descriptions, which often contain textual clues about bug types, (2) simultaneously processing the textual and code components of patches — something static analysis struggles with, and (3) new patch patterns can be relatively easily incorporated using the LLM's few-shot learning capabilities [25], in contrast to rigid symbolic rules.

Design strategy 2: Classifying patches using a local model. While LLMs are very good at classifying patches with clear indicators, our preliminary experiments find that the effectiveness decreases significantly when patches do not have sufficient hints. This motivates the collection of additional code context beyond the original code diffs in the patch. To this end, we developed a custom program slicing method

```
@@ -316,6 +316,11 @@ int
st21nfca_connectivity_event_received(struct nfc_hci_dev *
hdev, u8 host,
          return -ENOMEM;

      transaction->aid_len = skb->data[1];

+
+     /* Checking if the length of the AID is valid */
+     if (transaction->aid_len > sizeof(transaction->aid))
+        return -EINVAL;
+

      memcpy(transaction->aid, &skb->data[2],
             transaction->aid_len);
```

(a) A patch for an out-of-bounds access vulnerability

```
@@ -567,12 +567,11 @@ static void nfsd_init_dirlist_pages(
struct svc_rqst *rqstp,
   struct xdr_stream *xdr = &resp->xdr;

-  count = clamp(count, (u32)(XDR_UNIT * 2),
svc_max_payload(rqstp));
   memset(buf, 0, sizeof(*buf));

 /* Reserve room for the NULL ptr & eof flag (-2 words) */
-  buf->buflen = count - XDR_UNIT * 2;
+  buf->buflen = clamp(count, (u32)(XDR_UNIT * 2), (u32)
PAGE_SIZE);
+  buf->buflen -= XDR_UNIT * 2;

    buf->pages = rqstp->rq_next_page;
```

(b) A second patch for an out-of-bounds access vulnerability

Fig. 4: Examples of two common patch patterns for out-of-bounds access

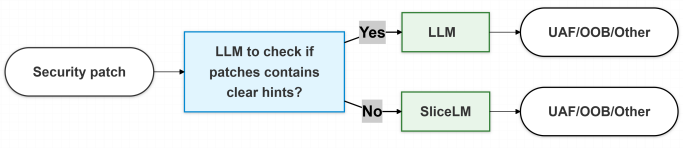

Fig. 5: The high level pipeline of DUALLM

that concisely captures the lines of code that constitute the bug, while minimizing the irrelevant lines (compared to the traditional slicing method). Unfortunately, since our custom slices are in a new, specialized data format, we find that the LLM cannot effectively utilize these slices even when we attempt to teach them via in-context learning (as shown later in § VI-A4). Thus, (given the unique nature of the data), we train a small language model, called SLICELM that can ingest such custom slices and infer the associated bug types.

Design strategy 3: A dual-method pipeline. Given the above, we categorize patches into two classes: those that contain clear vulnerability indicators (*patches with hints*) and those that require more code-level analysis (*patches without hints*). This fundamental distinction motivates the design of DUALLM, a dual-method pipeline that strategically handles each category with specialized techniques, as illustrated in Figure 5. Specifically, we design our pipeline to first differentiate the two categories of patches leveraging an LLM. Patches with hints are processed using methods that can effectively leverage their indicators, while patches without hints are analyzed by our dedicated model that focuses on careful examination of code modifications and the vulnerability context. In summary, as discussed below, an LLM excels at leveraging hints, but a dedicated model designed to understanding code context beyond code diffs is essential when such hints are absent. This dual-method architecture stands in contrast to prior approaches like CoLeFunDa [80] and TreeVul [49], which rely on a single model throughout; by strategically integrating two models, DUALLM more effectively leverages both high-level semantic cues and deep code context

IV. DUALLM DESIGN

In this section, we describe the two main components of DUALLM: LLM-based patch classification and SLICELM (our own custom model). For the former, we focus on the design of prompt strategies that guide the LLM. For the latter, we focus on our custom slicing technique and how it is used to train our model.

Our approach assumes that each patch corresponds to a single vulnerability — an assumption shared by prior work [49], [70], [80]. This is further supported by the official Linux documentation [20], which advises: “Solve only one problem per patch.”, and is consistent with our empirical observations.

A. LLM-based patch classification

We have two high-level goals when we leverage the LLM for our task. First, we aim to exploit an LLM to extract hints from patches that have them (Design strategy 1). Specifically, we look for hints in (1) the natural language description of commit titles and messages and (2) common patch patterns observed in commit diffs for bug types of interest, such as use-after-free (UAF) and out-of-bounds (OOB) access. Second, we seek to differentiate patches with and without hints, so that we can feed them to the corresponding methods in our pipeline (Design strategy 3).

In our design, we focus on guiding an LLM to effectively identify vulnerability indicators through both commit descriptions and common patch patterns. If successful, the bug type will also be reported and extracted subsequently. Otherwise, they are fed to our dedicated model. LLMs are known to be extremely effective at processing natural language; with respect to patch patterns, in contrast to SID [70] which uses precise but rigid symbolic rules to identify them, we hypothesize that LLMs can identify patch patterns in a more flexible way, e.g., recognizing small variations of the same underlying pattern. To this end, we encode common patch patterns in the form of examples for few-shot learning [25].

Patch pattern encoding (in-context learning). The downside of not relying on precise symbolic rules is that it also creates room for misclassification, especially when ambiguities arise. To overcome this challenge, we present both UAF/OOB

examples and non-UAF-OOB examples as guidance for the LLM in the form of few-shot prompts [25]. Specifically, our selected non-UAF-OOB examples cover various memory corruption bug types that involve changes to **critical memory operations** (e.g., initialization, free, and use). Since UAF and OOB are both related to vulnerable memory operations, these chosen patches share more similarities with UAF and OOB compared to patches that do not involve vulnerable memory operations. **The overlaps of repair characteristics** in terms of vulnerable memory operations can cause confusion in a classification task. For example, patches addressing both UAF and memory leak bugs often involve memory deallocation operations, such as `free()`. A memory leak bug is typically fixed by adding a `free()` somewhere. A UAF bug may be patched by deferring a `free()` [12] — both may exhibit a line with added `free()`. These examples are not enforced as hard-coded decision rules (e.g., as in SID [70]). Instead, the LLM uses these examples inductively to generalize across diverse real-world patches. This enables **greater flexibility and adaptability**, especially when handling variations in patch style, naming, or structure. Thus, while DUALLLM incorporates human-defined examples, it avoids fixed symbolic rule logic, and relies instead on the LLM's ability to reason and adapt.

By feeding explicit example patch patterns for these memory corruption bug types, we seek to help the LLM differentiate them better. Specifically, the patterns we encode for *different memory corruption bug types* are as follows:

- *out-of-bounds access*: add boundary check; recalculate memory area sizes.
- *use-after-free*: nullify pointers after freeing.
- *null pointer dereference*: add null pointer validation checks.
- *use before initialization*: add proper variable initialization.
- *memory leak*: insert memory free function calls along an execution path.

For the above patch patterns, our prompts follow the chain-of-thought strategy [67] by providing both representative patches and detailed explanations of how to interpret them and identify key indicators — see the last two boxes in Figure 6 which illustrates our prompt template. This approach guides an LLM through the intermediate reasoning steps needed to analyze patches, helping it decompose complex patch analysis tasks into manageable steps and ultimately improve its classification accuracy.

Note that our approach can be easily extended with more patch patterns by few-shot learning to provide examples in prompts. To demonstrate this, we expand the patch patterns for OOB vulnerabilities beyond what was supported by SID. Specifically, while SID only accounts for the pattern of "adding bounds checks", our approach in addition incorporates "recalculating memory area sizes" as a supported pattern. Extending the patterns and vulnerability types beyond what we have currently remains a direction for future work we can explore.

Prompt template. In Figure 6, we present the structured

I want you to act as a Linux kernel security patch expert, who is great at analyzing Linux security patches.

... I need your help to tell me if its commit title, message and diffs can contain reliable hints to tell us its type of the bug fixed by the patch. I will give you commit title, commit message, commit diff and three lines around diff as context. I hope that you can first look at commit title and commit message, since those descriptions can contain explicit or implicit hints about the bug type; and then commit diff and its context, which can fall into some common patch patterns(listed below). There are three bug types that I care about: use-after-free, memory out-of-bounds, and non-uaf-oob(including null pointer dereference, use before initialization, memory leak, and others).

Below are examples that help you learn this:
/*One of six patch examples, others not shown for brevity*/
Commit title: NFSD: Protect against send buffer overflow in NFSv2 READDIR
Commit message: Restore the previous limit on the @count argument to prevent a buffer overflow attack.
Commit diff:

```
@@ -567,12 +567,11 @@ static void nfsd_init_dirlist_pages(struct svc_rqst *rqstp,
  struct xdr_buf *buf = &resp->dirlist;
  struct xdr_stream *xdr = &resp->xdr;
- count = clamp(count,(u32)(XDR_UNIT * 2),svc_max_payload(rqstp));
   memset(buf, 0, sizeof(*buf));
- buf->buflen = count - XDR_UNIT * 2;
+ buf->buflen = clamp(count, (u32)(XDR_UNIT * 2), (u32)PAGE_SIZE);
+ buf->buflen -= XDR_UNIT * 2;
   buf->pages = rqstp->rq_next_page;
   rqstp->rq_next_page++;
```

Analysis:
(1) Commit title says it fixes a buffer overflow
(2) The patch diff changes the calculation of buf-buflen, which is the length of the buffer. It matches the memory size resetting patch pattern for memory-out-of-bounds.
Contains reliable hints about the bug type: yes;the bug type is memory out-of-bounds access.

...Be careful: if you are not 100% sure about if there are reliable hints about the bug type, you should say no; if there are conflicts between commit descriptions(commit title and message) and commit diff, please prioritize commit diff, and if commit diff can not provide certain hints, you should say no.
Please analyse it step by step according to the above methods.
Here is the commit that I would like you to analyze:

Fig. 6: Structured prompt template for patch classification; the blue boxes delineate, in order, five key components: (1) role specification, (2) task description, (3) example, (4) analytical methodology, and (5) extra guidelines.

prompt template for LLM-based patch classification. As one can see, we structure the prompts into five parts. First, we establish the role specification, defining an LLM as a Linux kernel security patch expert. Second, we provide a detailed task description that outlines the analysis requirements, including the input format and the goal of the task. Third, we include an example (one of six patch patterns we support, others omitted for brevity) to demonstrate the expected analytical approach. Fourth, we present the analytical methodology, showing step-by-step reasoning that covers both natural language descriptions and code diff pattern interpretation. Finally, we emphasize the importance of certainty in classification.

It is worth noting that towards the end of the prompt, we explicitly ask the LLM to "give up" on the classification if there are no reliable hints about the bug type. This is important

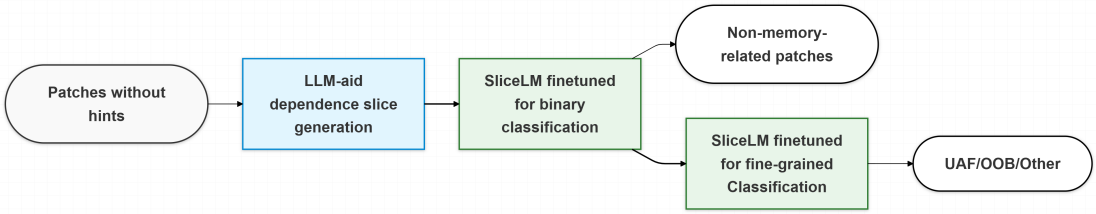

Fig. 7: SLICELM's pipeline to classify patches without hints

because we do not want to force the LLM to make an inference unconditionally; otherwise, the results may be inaccurate.

B. SLICELM-based patch classification

SLICELM is designed to classify **patches without hints**, i.e., patches that are classified by an LLM as such. As discussed in §III-B (design strategy 2), proper code context is critical to identify the type of bug being fixed in a patch and the context needs to be carefully extracted to capture the true nature and critical elements of the vulnerability. Specifically, the context must be designed to **capture the lines of code most semantically related**. Accomplishing this is no easy task. Relying on the default context of three lines around code diffs risks overlooking critical patch-relevant details, while using entire patched functions or traditional program dependence slices can confuse the model with irrelevant lines of code.

To solve this, we developed a custom slicing method. Given our slices are specialized, we choose to pre-train a BERT-based language model from scratch (details will be presented in §V), to grasp dependence-based relationships between slices and patch diffs. This is different from prior work [49] relying on the already pre-trained CodeBert model [32]. Then we fine-tune the model for two tasks: (1) the binary classification of memory corruption patches and other non-memory-related patches; and (2) the fine-grained classification of memory corruption patches into UAF, OOB and others. The workflow of SliceLM is presented in Figure 7.

1) Custom slicing: Slicing is an appropriate solution to extracting the code context (§III-B). Ideally, slicing should include all relevant code — but *only* the relevant code — to capture the critical operations tied to the bug's behavior. For example, we would like to find the lines of code that indicate a freed object being *used* subsequently, for a UAF bug. However, it is hard to determine how far away such operations are from the changed code lines in the code diff. If we arbitrarily increase the scope of a slice, e.g., making it inter-procedural, we would encounter significant noise from irrelevant code inclusion and face scalability challenges. To address this challenge, we develop three novel strategies: *(1) a selective slicing heuristic to prioritize data dependence over control dependence, (2) an LLM-aided function renaming method to enhance the vulnerability context while avoiding inter-procedural analysis; (3) an LLM-aided strategy to prune redundant code changes.* We detail the design of the three strategies next.

Selective slicing rule (avoiding bloating). We take the removed and added code of a patch as the slicing criterion, i.e., the starting point of slicing [68], because the variables changed in these lines are relevant to the bug. We observe that traditional program slicing often produces excessively large slices by including all data and control dependencies—even those unrelated to the bug, such as variables and operations that do not impact its presence or absence.

In particular, control dependence can lead to bloated slices. To illustrate, consider the example patch in Figure 8. The patch has introduced two additional lines of code that would return early if a specific condition is met. If we apply the standard forward control dependence slicing on the conditional statement, the slice would include all the lines after line 10. This is because whether these subsequent lines will be executed is control-dependent on the conditional statement, i.e., lines after 10 will not be executed if the new condition on line 9 is true. However, we know that the majority of the subsequent lines of code are not relevant to the bug. Instead, only the key operations directly involving the variable being checked (in this example `addr`) should be retained in the slice.

To mitigate such control-dependence-induced bloating, we redefine the handling of control dependence by introducing a focused, variable-driven slicing rule that balances precision and relevance. Instead of including all control-dependent statements, our approach selectively incorporates statements based on their direct interaction with the slicing criterion: *(1) forward control dependence slices include only statements that use variables involved in the criterion (e.g., conditional checks), and (2) backward control dependence slices include only conditional statements that use variables in the criterion. This hybrid approach blends aspects of control dependence and data dependence slicing, prioritizing patch-relevant impact while minimizing irrelevant code.* The intuition is to limit the inclusion of code due to control dependence to those that are more likely to contribute to understanding the impact of patches, e.g., checking a length variable or the validity of a pointer.

To illustrate our approach, let us look at the slices we obtain if we use line 9 of Figure 8 as the slicing criterion (since line 9 is an added line). Backward slicing yields lines 6 and 8 due to data dependence for variable `addr`. Traditional forward slicing will include all of the code after line 10 (more than 40 lines), since their execution will be controlled by the if-return check in lines 9 and 10. However, as discussed previously, our selective slicing will include only the lines which use

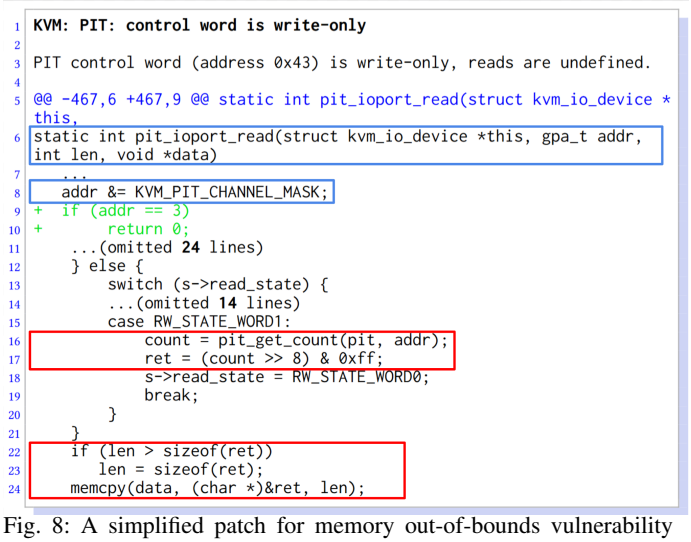

Fig. 8: A simplified patch for memory out-of-bounds vulnerability (**Blue** boxes indicate the extracted backward slices and **red** boxes indicate the extracted forward slices using our method)

the variable checked in the if condition; therefore, line 16 is included in the forward slice (which is directly related to patch semantics); line 17 is included since it uses the variable `count` defined in line 16; line 20, 21 and 22 are also included due to the use of variable `ret` defined in line 17. In other words, our forward slicing yields lines 16, 17, 22, 23, and 24, which captures all operations relevant to the OOB vulnerability and avoids significant noise compared to the 40+ lines present in the forward slice of the traditional method. Using the slice, we can see that the patch adds a check on a variable `addr`. This variable `addr` is an index that is used to retrieve an address `ret` and the size of source data to be copied. Without imposing constraints on `addr` (line 9-10), it is possible that the size of the source data is bigger than that of the destination data during `memcpy()`, causing a memory out-of-bounds access in line 24. These lines are necessary and sufficient for the ML model to make an inference that an OOB vulnerability is being fixed by the patch. If the traditional program slicing method is used, we observe that our model will misclassify this case.

LLM-aided slicing (function renaming). As shown in the example in Figure 8, obtaining complete vulnerability context often requires tracking operations that are distant from the code diff, e.g., `memcpy()`. In fact, it can even require tracking across function boundaries through inter-procedural analysis. However, performing inter-procedural slicing can be prohibitively expensive and may introduce substantial amounts of irrelevant code, potentially degrading the performance of the model we aim to train. Therefore, we opt for intra-procedural slicing, though this inevitably excludes relevant context from other functions.

To address this limitation, we leverage a key observation that function semantics are often revealed in function names: they typically provide clear indicators of their behaviors [46]. For example, the function `drm_mm_remove_node()` suggests its purpose of node removal. Since function names are included in the code diffs and slices, they should in principle already contain semantics that can benefit patch classification. However, the sheer diversity of functions and their names can be a hurdle in training our local model: many projects define wrappers or custom-named functions for memory operations like allocation, freeing, reference counting, or copy operations, using names that are not standardized or intuitive.

We leverage an LLM to interpret and analyze the functionality of open-source functions (e.g., in the Linux kernel) and convert them into **more standardized function names** to aid the training and classification process. This capability stems from LLMs being pre-trained on extensive datasets (including Linux source code and documentation), a finding also confirmed by prior work [43]. Specifically, we rename functions according to their relevant *memory operations* because these behaviors are closely tied to memory corruption vulnerabilities. Our approach asks an LLM to **identify and rename functions based on key memory operations** including allocate, free, read, write, map, copy, and reference count increase/decrease. When an LLM identifies that a function's primary functionality is one of these operations, it renames the function accordingly; otherwise, the original name is preserved. For instance, `l2cap_chan_hold_unless_zero()` is renamed to `increase_reference_count_if_not_zero()` to explicitly reflect its reference count operation. Finally, we replace original function names in custom slices and code diffs with their semantically renamed counterparts. As more patches are analyzed, we accumulate these mappings and cache them to minimize inference overhead. This enables the downstream model to reason about what a function does, rather than what it is called.

Standardizing function names semantically creates a unified vocabulary for memory operations, enabling our model to more effectively identify and correlate similar memory behaviors across different code segments. Considering the example in Figure 9, the UAF issue arises because `put_cred(creds)` is executed even if the variable `ret` is '1' (which is a valid return value), causing an unwanted reference count decrement and potentially triggering an unwanted free. Our function renaming approach assigned the name `decrease_credential_reference_count()` to the function `put_cred()`, emphasizing its connection to the reference counter decrement. Without this renaming, we found that our model would misclassify the case as OOB instead.

LLM-aided slicing (pruning redundant code changes). The above method is sufficient when we consider relatively small patches. However, large patches in the real world can contain numerous code changes spread across multiple functions and even files; `yet, not all changes target fixing the vulnerability.` Unfortunately, we cannot directly ask an LLM to pinpoint the bug-logic-relevant changes. This is because the patches we process at this stage are already considered "without hints", and pinpointing the bug-logic-relevant changes would require clues about the bug type.

```
io_uring: fix xa_alloc_cycle() error return value check

We currently check for ret != 0 to indicate error, but '1'
is a valid return and just indicates that the allocation
succeeded with a wrap. Correct the check to be for < 0, like
 it was before the xarray conversion.

@@ -9843,10 +9843,11 @@ static int io_register_personality(
struct io_ring_ctx *ctx)
    ret = xa_alloc_cyclic(&ctx->personalities, &id, (void *)
creds,
          XA_LIMIT(0, USHRT_MAX), &ctx->pers_next, GFP_KERNEL
);
-   if (!ret)
-       return id;
-   put_cred(creds);
-   return ret;
+   if (ret < 0) {
+       put_cred(creds);
+       return ret;
+   }
+   return id;
 }
```

Fig. 9: An example where function renaming helps

That said, there are still opportunities to prune clearly redundant code changes in a bug-agnostic way. Specifically, we propose to leverage an LLM to identify two patterns of redundant code changes: *(1) redundant changes (same code changes that appear in multiple places due to the change of function call parameters), and (2) code refactoring without changing the actual code behaviors.*

```
@@ btrfs_ioctl_resize:
-device = btrfs_find_device(fs_info->fs_devices, devid, NULL
, NULL);
+device = btrfs_find_device(fs_info->fs_devices, devid, NULL
, NULL, true);

@@ btrfs_scrub_dev:
-dev = btrfs_find_device(fs_info->fs_devices, devid, NULL,
NULL);
+dev = btrfs_find_device(fs_info->fs_devices, devid, NULL,
NULL, true);

(11 other instances of changes similar to the above)

@@ device_list_add:
-device = find_device(fs_devices, devid,disk_super->dev_item
.uuid);
+device = btrfs_find_device(fs_devices, devid,disk_super->
dev_item.uuid, NULL, false);

-find_device() // function is totally removed

+btrfs_find_device() // add a bool argument and
corresponding operation for it

 ...
```

Fig. 10: A simplified patch for redundant changes example; Blue boxes are semantically equivalent, red boxes are critical lines

Figure 10 illustrates an example with many change sites that are basically redundant. The patch is composed of **17** change sites (for brevity, we only show 6). However, the first **13** change sites are semantically equivalent, since all of them are adding a `true` argument when calling `btrfs_find_device()`, as shown in the first two change sites highlighted in the two blue boxes in Figure 10. We find that the critical lines for the patch are unrelated to *any* of repeated instances, e.g., the code within blue boxes in Figure 10 that is repeated. Including all these "similar" changes and their corresponding dependence-based context would not only obscure the core vulnerability-fixing modifications (the meaningful change is highlighted in the red box) but also consume valuable space in our model's limited input length (1,024 tokens). The LLM provides us with an effective means of identifying such cases and preserving only a single representative instance of multiple equivalent code modifications.

```
//implement a new function called xfs_iget_check_free_state() mainly
 composed of if checks
//for brevity, we only show one 'if check' below
+       ...
+       if (ip->i_d.di_nblocks != 0) {
+           xfs_warn(ip->i_mount,
+"Corruption detected! Free inode 0x%llx has blocks allocated!",
+               ip->i_ino);
+           return -EFSCORRUPTED;
+       }
+       ...

@@ xfs_iget_cache_hit:
+call xfs_iget_check_free_state() and check return value

@@ xfs_iget_cache_miss
// remove 'if checks' that are added to xfs_iget_check_free_state()
now
// for brevity, we only show one 'if check' below
-       ...
-       if (ip->i_d.di_nblocks != 0) {
-           xfs_warn(mp,
-"Corruption detected! Free inode 0x%llx has blocks allocated!",
-               ino);
-           error = -EFSCORRUPTED;
-           goto out_destroy;
-       }
-       ...
+call xfs_iget_check_free_state() and check return value
```

Fig. 11: A simplified patch for code refactoring example

Figure 11 depicts a simplified example where the changes highlighted in the blue box represent a modification that does not alter the underlying semantics of the code — it merely relocates "if" checks into a new function call. For such refactoring, we ask an LLM to identify and remove such added and deleted code segments. After pruning, it is clear that the patch in Figure 11 mainly added additional checks for `xfs_iget_cache_hit()`, as shown in the second change site. This selective preservation produces a concise input for our model (SLICELM) by retaining only essential code modifications and discarding irrelevant or redundant ones.

2) Two-stage classification for patches without hints: We fine-tune the pre-trained model for two downstream tasks, which are two classifiers, as shown in Figure 7. *The first classifier is a binary one, classifying the patches into memory corruption vs non-memory-related.* As noted in §IV-A, the memory-related bug types we consider include: use-after-free, memory out-of-bounds access, use-before-initialization, null pointer dereference, and memory leak. This is motivated by the observation that memory corruption patches inherently exhibit distinct characteristics from other types – they tend to involve memory operations such as `free()` and `memcpy()`. Successfully ruling out non-memory-related patches can help improve the accuracy of the subsequent classifier (as will be shown in § VI-A4). *The second classifier then distinguishes the patches into UAF, OOB, and other memory-corruption patches.*

V. Implementation & Experiment setup

A. Implementation Details

LLM usage. Our experiments were conducted primarily using OpenAI GPT-4-turbo [9]. In addition, we perform experiments using the Llama 3.1 405B [16] and OpenAI-O1 [17].

Custom slicing. We implemented our custom slicing at the source code level, using Joern [74], [13]. The slice is computed for each change site, which includes the added lines and/or removed lines (with the plus and minus signs preserved) as well as the forward and backward slices.

Additional information besides slices. In addition to the custom slices, we also feed a single-sentence summary of the patch description to our model. The summary is generated by leveraging LLM's summarization capabilities to distill verbose commit messages into a concise sentence. The reasoning is that commit messages often contain useful information, e.g., about how a variable is used. Even if it does not directly hint at the bug type, it can potentially provide additional information when combined with slices. Nevertheless, feeding long commit messages to our local language model consumes significant token space, making summarization necessary.

B. Experiment Setup

The pre-training of the model typically took approximately six days. On average, the classification of a test case was achieved within 30 seconds (via the entire pipeline). Further details about model training (e.g., model architecture, optimization settings) are provided in Appendix A.

Pre-training dataset. SliceLM is pre-trained on a comprehensive dataset of all available historical commits since the inception of the Linux kernel repository up to kernel v6.0, encompassing 1.1M commits. This process enhances the model's capability to grasp dependence-based relationships between slices and patch diffs, thereby capturing semantic features within the source code. We subsequently perform two separate fine-tuning processes on the pre-trained model: one for binary classification and the other for multi-class classification.

Fine-tuning dataset. We prepare two datasets, one for each of the two fine-tuned tasks. For the multi-class classification, we select patches with standardized phrases in commit titles that indicate the type of the fixed bug to collect ground truth. For example, phrases like "Fix use-after-free" or "Fix out-of-bounds" suggest the type of the fixed vulnerabilities. We compiled **a list of common key phrases** (show in Appendix B) used in the Linux kernel community and mapped them to the three bug types. This key-phrase-based labeling strategy generated a dataset of 10,540 samples.

To create the memory corruption and non-memory-related datasets, we used the above samples as positive samples (related to vulnerable memory operations) and commits labeled as "non-security" in `PatchDB` [64] as negative samples. It is worth noting that finding non-memory-related security patches is challenging, as there are many types of vulnerabilities which are difficult to enumerate. We believe the non-security patches in `PatchDB` provide a reasonable approximation of non-memory-related security patches, since non-security patches are also technically free of any vulnerable memory operations.

Evaluation dataset #1: CVE patches. Our first evaluation is on a quality-controlled dataset of Linux kernel CVE patches. Since CVE patches are, by definition, confirmed security patches, we do not need to apply a binary classifier to distinguish between security and non-security patches. Instead, we can directly evaluate fine-grained security patch classification methods, e.g., such as DualLM and TreeVul. This allows a fair comparison, as all evaluated methods operate on the same set of established security patches. Please note that this is akin to taking the output of an "ideal binary classifier" and feeding the same as the input to a fine-grained classifier (the latter being the focus of our effort). If a non-ideal binary classifier is used (wherein it does not yield a 100% accuracy in classification results either due to improper training sets, overfitting or otherwise), the results will negatively influence the performance of the overall pipeline but is not directly attributable to the performance of the fine-grained classifier. This collected CVE dataset consists of 946 patches manually assigned CWE labels, from 2015 to May 2024. They include 78 CVEs published in 2024, which is after the knowledge cutoff date of the LLM we used in the evaluation. This can help us understand the potential data leakage problem where an LLM might gain an advantage because it might already have learned the details about the older CVEs in training data. Note that if evaluated cases occur in the fine-tuning dataset, they are excluded from the dataset during training to prevent overlap. According to CWE labels, we mapped these CVE patches to OOB, UAF, and non-UAF-OOB, by manually verifying the labels. More details of the data cleaning can be found in Appendix C). Since this dataset is **quality-controlled and complete on the labels,** the goal of the evaluation is the comprehensive evaluation of classification performance, enabling both a systematic comparison with state-of-the-art approaches and detailed ablation studies. *We present the results of this evaluation in in § VI-A.*

Evaluation dataset #2: Unlabeled Patches. Second (and perhaps more critically), we challenged DualLM with 5,140 Linux kernel patches to assess its ability to discover previously unknown UAF and OOB bugs — a test that mirrors the actual deployment scenarios where automated patch classification can prevent potential security breaches. Note that DualLM is designed to operate on security patches as input (same with TreeVul). Therefore, to evaluate its performance in a realistic end-to-end setting—where the security relevance of a patch may not be known in advance—we combined it with various upstream binary classifiers (security vs. non-security). This enables an assessment of how well the full pipeline performs in practice, from initial security identification to fine-grained classification. Differently, SID does not require the input to be security patches. These patches were selected randomly from a total of 12K patches on the Linux kernel Long-Term Support (LTS) version 6.6 branch, as long as their patch date

was in 2024 (after the knowledge cutoff date of the LLM we used). We choose these patches regardless of whether they are assigned CVE labels. *The results of this evaluation are presented in § VI-B.*

Comparison setup. We choose TreeVul [49] and SID [70] as comparison targets since they are the state-of-the-art open-source solutions that share similar goals with our work. For TreeVul, since it classifies patches into CWE labels (including types beyond OOB and UAF), we mapped these labels to our three target categories: OOB, UAF, and non-UAF-OOB, consistent with our classification scheme. Furthermore, to ensure fairness and avoid data leakage, we excluded the CVE patches present in TreeVul's original training dataset from our CVE-based evaluation dataset. Note that we do not directly compare with CoLeFunDa [80], because we are unable to obtain its source code.

For SID, which employs a rule-based matching method to specifically identify use-after-free, out-of-bounds, null pointer dereference, and use-before-initialization bugs, there is no explicit category corresponding directly to our non-UAF-OOB classification. To align SID's outputs with our categories, we adopted the following approach: 1) patches matched by SID are classified into their corresponding bug types (UAF or OOB), and 2) unmatched patches, as well as patches matched as null pointer dereference or use-before-initialization, are grouped into non-UAF-OOB. Since SID uses rule-based matching rather than machine learning, it does not require training data; therefore, our evaluation dataset for SID includes the entire CVE dataset without exclusions.

Manual verification. To support the evaluation of our 5,140 real-world patches in terms of ground-truth validation, we perform targeted manual verification, as described below. Labeling all 5,140 patches is an extremely time-consuming and labor-intensive task, making full manual annotation infeasible. To balance rigor and scalability, we adopt a tiered verification strategy: (A) We manually verify all UAF and OOB cases identified by our pipeline using a structured multi-step process grounded in kernel security expertise (described below and illustrated in Figure 12); (B) For patches classified as non-UAF-OOB, we sample 50 cases for manual review to estimate the false negative rate; (C) We select representative cases to develop proof-of-concepts (PoCs) and real exploits to further validate classification correctness. With respect to (A), our multi-step process is as follows: 1) **Commit Message Inspection**: We first check whether the commit title/message contains explicit or implicit hints. Phrases like "fix use-after-free" are considered reliable indicators, as kernel developers generally document key issues therein. 2) **Locate Vulnerable Variable**: We examine code changes. With respect to OOB, we look for signals such as bounds checks, size clamping, array index corrections, in order to locate the possible vulnerable variable. Next, we manually trace across function boundaries whether the modified variables are involved in memory accesses, such as indexing or `memcpy()` operations. With respect to UAF, we identify signals such as pointer nullifications after free, reference count adjustments, introduction of locks, or reordering of deallocation logic, to locate the possible vulnerable variable. We also review whether a function indirectly performs a free (via wrappers), and whether the code adds or delays such a free. 3) **Reconstruct the Vulnerability Logic**: For complex or ambiguous cases, we reconstruct the calling context based on clues in the commit message or associated stack traces (if available). We trace data flows and call chains across functions to identify the vulnerable logic path and verify whether the bug could lead to memory corruption. 4) **Conservative Labeling**: We assign labels only when confident. Uncertain cases are categorized as non-UAF-OOB to ensure dataset integrity. Five analysts with security expertise independently follow this procedure. In cases of disagreement, the analysts discuss the patch collectively and reach a consensus through majority agreement. On average, the manual analysis of a single case by one analyst takes approximately 20 minutes, highlighting the depth of inspection involved.

Fig. 12: The manual verification process for real world patches

Metrics. For most experiments, we report accuracy, precision, F1-score, false positive rate, and false negative rate. For multi-class classification evaluations, metrics such as precision can be more challenging to interpret, as multiple types of errors (e.g., predicting class A when it is actually class B, C, or D) must be accounted for. Therefore, we use the commonly adopted *weighted averaging* method [24], to assess the overall model quality in performing multi-class classification tasks. The method essentially calculates metrics like *precision*, and *F1-score* individually for each class, and subsequently aggregates a weighted average by assigning weights to all classes according to their frequencies.

VI. Experimental Results

A. Evaluation on Quality-Controlled CVEs

1) Main Results: Table I demonstrates DualLM's effectiveness, with an overall **87.4%** accuracy, **87.7%** precision, and a **0.875** F1-score, while maintaining a low false positive rate of **5.4%** and a relatively low false negative rate of **10.0%**. Since our solution is a dual-method pipeline, we also break the results down by method. We find that 77% of the evaluated patches are classified as patches with hints and 23% are classified as patches without hints. For patches considered with hints or indicators, DualLM achieves a 90.3% accuracy with a 90.7% precision and a low false positive rate of 4.0%. Even for those without indicators, our two-stage classification approach maintains high effectiveness, achieving an overall 77.9% accuracy. Note that although the binary classification and fine-grained classification achieved 84.8% accuracy and 81.8% accuracy respectively, their combined accuracy is lower because errors from the two stages compound when the results are composed. In summary, we believe these results are

significant given the challenging nature of patch classification, especially when compared against the state-of-the-art, which we describe next.

2) Comparative study: To further validate DUALLM's effectiveness, we compare its performance against that of two open-source, state-of-the-art solutions, viz., TreeVul [49] and SID [70]. The comparison setups are described in §V-B.

In Table II, we show the comparison with TreeVul and see a substantial performance gap: TreeVul achieved only a **65.6%** accuracy with an F1-score of 0.653, while DUALLM achieved significantly better performance with a **86.8%** accuracy and an F1-score of 0.869. Upon examining some specific cases, we identify two major reasons for the performance discrepancy: (1) misclassified cases by TreeVul actually have indicators in the commit titles and/or messages, which allows DUALLM to correctly classify them; (2) misclassified cases by TreeVul needs the proper context to infer the correct type. TreeVul only takes code diffs as input, ignoring the commit title/message and code context.

Figure 9 illustrates an interesting example where TreeVul failed to identify the bug type. As discussed in § IV-B1, our function renaming, which gave the new name `decrease_credential_reference_count()` for `put_cred()`, plays an important role in the correct identification of the associated UAF bug. However, TreeVul does not appear to capture the knowledge about what `put_cred()` does (perhaps due to limited training data). In addition, we suspect that TreeVul's automatic alignment mechanism actually makes it pay less attention to `put_cred(creds)` because the function is present in both the pre-patch and post-patch version.

Table III shows the comparison with SID, and the results demonstrate a similar performance gap: SID achieves an accuracy of **63.8%** and an F1-score of 0.546, compared to DUALLM's **87.4%** and 0.875, respectively. Importantly, SID's false negative rate is alarmingly high at 67.8%, meaning that it misses more than half of the vulnerabilities. In contrast, DUALLM achieves a much lower false negative rate of 10.0% while also reducing the false positive rate to 5.4% (compared to SID's 21.2%).[2]

One might question whether DUALLM 's use of multiple few-shot examples, including those outside the scope of SID (e.g., "memory area size recalculations"), gives it an unfair advantage. To address this, we repeated the evaluation with a constrained version of DUALLM using only one OOB patch example that matches the patch pattern used by SID (i.e., boundary check additions). The result only degraded slightly and remains strong: DUALLM achieves an accuracy of 86.9% and an F1-score of 0.869, still significantly outperforming SID.

Our analysis shows that SID's effectiveness is limited by its reliance on symbolic pattern matching rules, which support only one pattern per vulnerability type. For instance, in detecting out-of-bounds vulnerabilities, SID looks exclusively for boundary check additions, missing other common fixes like memory area size adjustments or more nuanced protection mechanisms (as illustrated in Figure 8).

3) Generalization to other bug types and beyond Linux: Although we have focused on the most severe types of bugs, i.e., UAF and OOB, we have also evaluated DUALLM's generalizability by training/testing it to classify other bug types. For this expanded experiment, the binary classification step ('memory corruption vs. non-memory') requires no retraining. However, the fine-grained classification step does require retraining. Specifically, we reuse the same pretrained model and apply the same labeling method described in § V-B to assign fine-grained labels (e.g., use-before-initialization, null pointer dereference) beyond the original OOB and UAF labels to the same train set of 10,540 patches. When expanding to a broader set of vulnerability types, including memory out-of-bounds access, use-after-free, null pointer dereference, use-before-initialization, memory leak and non-memory-related bugs, DUALLM maintains robust performance with an **80.35%** accuracy and an F1-score of 0.801. This demonstrates our method's ability to handle **diverse vulnerability types** while maintaining high classification accuracy.

In addition, even though we focused on the Linux kernel due to its vast scale and complexity, we also tested two other open-source projects: FFmpeg [7] and OpenSSL [18]. Specifically, we followed the same data collection and labeling methodology described in § V-B: we collected CVE patches for these projects by retrieving entries from the corresponding CVE database, automatically mapping their associated CWE labels into our bug type taxonomy, and then applied the same verification process to correct any mislabeled or ambiguous cases. The results are equally compelling, with DUALLM achieving F1-scores of 0.832 and 0.834 respectively, demonstrating its **generalizability across software**.

4) Ablation Study: The effectiveness of DUALLM stems from several key design strategies. Through an ablation study, we now demonstrate how these design choices contribute to the overall performance.

Dual-method design. Our first experiment targets understanding whether the LLM-based method or SLICELM alone can achieve good results by themselves (similar to that of DUALLM). To test the LLM-only solution, we adapt the original LLM-based classification to always force an answer, instead of allowing the LLM to defer the answer when it deems that there are insufficient indicators in the patch. To test the SLICELM-only solution, we simply feed all the patches to SLICELM without any modifications to the model. Table IV shows the results. Interestingly, both the LLM-only and SLICELM-only solutions yield promising results, with similar performance across metrics such as accuracy, precision, etc. *Notably both already outperform SID and TreeVul, but are inferior to* DUALLM.

Combining the two methods yield improved results across the board, e.g., an **11%** improvement in overall accuracy.

[2]SID's performance on our evaluation dataset appears significantly lower than what was reported in the original paper. We consulted the SID authors when we ran SID against the evaluation dataset to ensure that we ran it correctly. One possible reason is the difference between the two evaluation datasets. The 97% precision reported in the SID paper is on a different dataset than ours (we used the CVE dataset, including many patches that were published after the SID paper was published).

Task	Accuracy	Precision	F1-score	FP Rate	FN Rate
Overall pipeline	**87.4%**	87.7%	0.875	5.4%	10.0%
– **Classification on patches with hints**	**90.3%**	90.7%	0.904	4.0%	6.6%
– **Classification on patches without hints**	**77.9%**	78.4%	0.776	13.4%	24.1%
| Binary classification	84.8%	85.8%	0.847	13.9%	16.5%
| Fine-grained classification	81.8%	83.5%	0.821	8.8%	18.8%

TABLE I: The performance of DUALLM

Method	Accuracy	Precision	F1-score	FP rate	FN rate
TreeVul	65.6%	72.5%	0.653	21.6%	23.0%
DUALLM	86.8%	87.5%	0.869	5.1%	9.3%

TABLE II: Comparison between TreeVul and DUALLM.To ensure a fair comparison and prevent data leakage, we excluded the CVE patches used in TreeVul's original training dataset

Method	Accuracy	Precision	F1-score	FP rate	FN rate
SID	63.8%	77.7%	0.546	21.2%	67.8%
DUALLM	87.4%	87.7%	0.875	5.4%	10.0%

TABLE III: Comparison between SID and DUALLM

These results demonstrate the complementary nature of the two methods. Particularly noteworthy is DUALLM's improvement on false positives and false negatives simultaneously. It reduces the false positive rate to 5.4% (compared to 12.6% for LLM-only and 10.1% for SliceLM-only) and the false negative rate to 10.0% (versus 19.0% and 16.4% respectively).

Context options. Code context is particularly crucial for the classification of use-after-free, memory out-of-bounds access and other memory corruption patches since their differences are subtle, as discussed previously. Thus, our ablation study on context options is done for this challenging fine-grained classification task (the second stage of our two-stage classification). Table V depicts the results. The three lines before and after patch diffs as code context (the conventional and default way [8]) proves inadequate, achieving only **54.5%** accuracy. Traditional program dependence based slices show a moderate improvement to yield an accuracy of **58.1%**, but still fall short of capturing crucial vulnerability patterns.

Our custom slicing consists of three components: selective slicing, function renaming, and pruning of redundant code changes. Here, we seek to understand their individual and cumulative contributions. Selective slicing alone significantly improves performance over traditional program slicing, achieving 67.0% accuracy compared to 58.1%, thanks to improved dependency selection and noise reduction. Adding function renaming to selective slicing further improves performance to 78.0% accuracy and an F1-score of 0.789. This confirms that normalizing semantically equivalent memory-related functions improves the model's ability to generalize across diverse patches. Finally, our full slicing pipeline—which includes selective slicing, function renaming, and pruning of redundant code changes—achieves 81.8% accuracy with an F1-score of 0.821. Notably, the complete pipeline reduces the false positive rate from 16.5% (with selective slicing) to 8.8%, and the false negative rate from 34.1% to 18.8%. These results demonstrate that each component meaningfully enhances the effectiveness of code context analysis and collectively enables the model to better distinguish subtle variations in memory corruption patches. Note that these results correspond to the performance of SLICELM and not the entire DUALLM pipeline.

Method	Accuracy	Precision	F1-score	FP rate	FN rate
LLM-only	76.7%	81.1%	0.776	12.6%	19.0%
SliceLM-only	76.0%	80.0%	0.769	10.1%	16.4%
DUALLM	87.4%	87.7%	0.875	5.4%	10.0%

TABLE IV: Comparison between LLM-only, SliceLM-only and DUALLM

Method	Accuracy	Precision	F1-score	FP rate	FN rate
3-line contexts	54.5%	60.1%	0.531	21.5%	39.3%
Standard slices	58.1%	72.1%	0.635	29.1%	56.0%
Selective slices	67.0%	70.5%	0.680	16.5%	34.1%
Selective slices+ function renaming	78.0%	81.5%	0.789	10.5%	22.5%
SliceLM slices	81.8%	83.5%	0.821	8.8%	18.8%

TABLE V: The performance of SLICELM with different context options

Pass better context to the LLM. Now that we have demonstrated the importance of the right code context, a natural question arises: Can the LLM achieve performance similar to DUALLM if given access to our custom slices? To test this, we provide the LLM with carefully crafted prompts and examples demonstrating how to leverage the slice information. Interestingly, it achieves only a **62.6%** accuracy, with an F1-score of 0.664. This stands in stark contrast to SLICELM's superior performance of 81.8% accuracy and 0.821 F1-score. We conjecture that this is due to the LLM not being familiar with our custom slice format, which does not always conform to well-formed programs. This is in contrast to SLICELM, which is pre-trained and fine-tuned on our specialized slices.

To circumvent this issue, we perform another experiment wherein we feed the entire function as code context to the LLM. In theory, the LLM should automatically extract the needed information from the whole function. The results were better, but with only a **73.1%** accuracy and an F1-score of 0.732. Thus, there is still a gap compared to SLICELM's 81.8% accuracy and 0.821 F1-score. We suspect that this is because the markedly larger code context (whole functions) introduces noises and disrupts the LLM's understanding of a patch.

Two-stage classification for patches without hints. To validate the effectiveness of the two-stage classification of patches without hints, we compare the approach with a single-stage classification where we directly classify patches into OOB, UAF, and others, without first classifying them into memory corruption vs. non-memory ones. While direct one-stage classification achieves a **71.4%** accuracy with a 77.2% precision and a 0.738 F1-score, our two-stage approach sig-

nificantly improves performance to **77.9%** accuracy, 78.4% precision, and a 0.776 F1-score. These results demonstrate that decomposing the classification task into smaller sub-problems improves accuracy.

Using other LLMs. To demonstrate that our approach is not dependent on a specific LLM, we also evaluated DUALLM by using the popular open source Llama 3.1 405B. Overall, on the same testing dataset, we find that the overall pipeline still maintains strong performance, achieving **85.0%** accuracy, 85.0% precision with an F1-score of 0.849. It shows only a modest decrease from DUALLM with GPT-4-turbo. A detailed analysis reveals interesting differences between the models. Notably, when using identical prompts, Llama classifies fewer patches into the category with hints–65% versus 77% with GPT-4-turbo. Yet, its accuracy on such patches is only 88.0% compared to 90.3% in GPT-4-turbo. This indicates that Llama is more conservative in identifying patches with clear indicators. Interestingly, Llama defers more patches without hints to SLICELM, but SLICELM can still achieve 79.4% accuracy in these cases, which is higher than the result when GPT-4-turbo is employed (77.9%). We also evaluated DUALLM using the OpenAI-O1 model still using identical prompts. However, due to its high API cost, we only test 100 cases. In these cases, the overall performance is a **89.0%** accuracy, a 89.3% precision with a 0.890 F1-score. As with GPT-4-turbo, 23% of the cases are classified as patches without hints, and SLICELM achieves a 87% accuracy. 77% of the cases are classified as patches with hints; OpenAI-O1 has an 89.6% accuracy with respect to these. Overall, these results show that while the model choice affects specific performance metrics, our approach remains effective across other LLMs, validating its robustness and generality.

5) Robustness of Function Renaming: A potential concern is that function renaming based on key memory semantics may suffer from 'semantic drift' across different kernel versions; for example, a function may increase a reference counter in one version but not in another. To assess this, we manually analyzed 100 randomly selected functions from the Linux kernel, comparing their key memory semantics in two versions spanning five years: v4.19 (October 2018) and v6.6 (October 2023). We found that only 3% of the functions exhibited substantial semantic changes affecting memory-related behavior (e.g., transitioning from not freeing memory to including a `kfree()` operation). In contrast, 37% showed no memory-related changes, 26% involved only minor modifications (such as temporary variable refactoring) that do not affect semantic labeling, and 34% were absent in one version due to removal or major API changes. These findings suggest that semantic drift poses minimal risk to our renaming strategy in practice.

6) Analysis of the misclassified cases: It is inherently challenging to precisely interpret the failures of machine learning models. However, we make some observations on the cases in which DUALLM fails, in an attempt to contemplate how our solution can be further improved. For patches with hints (analyzed by the LLM component), nearly all misclassifications arise from inaccuracies in determining whether patches contain reliable hints. This can be attributed to hallucination phenomena in LLMs, where the model either incorrectly perceives or overlooks critical indicators. Specifically, two distinct error patterns emerge: (1) patches lack clear signals about the bug type and necessitate deeper code context analysis, yet the LLM mistakenly perceives that sufficient hints are present, leading to incorrect inference; (2) patches explicitly or implicitly contain hints about the bug type, but the LLM fails to accurately extract or interpret these signals. For patches without hints (analyzed by the SliceLM component), more than 50% of the misclassified cases exhibit complex structures in the patches. Some patches involve extensive modifications, such as removing a specific feature with hundreds of lines of code (sometimes spanning multiple files), and the vulnerable code is mixed in and removed at the same time. In other cases, the key logic that helps us understand the vulnerability type and the actual patch location are separated by multiple function calls in the program's call graph, requiring a far larger analysis scope than what we can capture (even with the help of function renaming). Interestingly, about 10% of the misclassified patches occur in cases where, from a human analyst's perspective, the patch diffs and slices contain sufficient information to determine the vulnerability type; yet, the model fails to make the correct classification. These cases highlight the gap between human and machine comprehension of patch features. The remaining failures stem from various other factors that we are unable to fully identify or systematically categorize.

7) Data leakage concerns with LLM: To address potential concerns about LLM performance being influenced by exposure to CVE patches during pre-training, we compared the results on patches before and after the LLM's training cutoff date (2024). On pre-2024 patches, DUALLM achieves 87.9% accuracy, and a 0.880 F1-score. For patches from 2024, it achieves 82.1% accuracy, and a 0.820 F1-score. While there is a moderate drop in performance, DUALLM still maintains reasonably strong performance, suggesting its effectiveness stems from its fundamental design rather than memorization of training examples. The system can generalize to unseen patches while maintaining robust performance.

B. Evaluation on Unlabeled Recent Kernel Patches

1) Comparative study: Besides the CVE patches, we also evaluated 5,140 recent patches collected from the Linux kernel 6.6 LTS branch in 2024 (details in §V-B). These patches include both security and non-security ones. Because DUALLM requires security patches as input, we evaluated it in combination with different upstream binary classifiers. Specifically, we used GraphSPD [62] and VulFixMiner [81] to identify security patches from 5,140 recent patches. As for PatchRNN, we found it unsuitable for this task. It labeled nearly 97% of test cases as "security." We manually verified 50 cases and found that from these predicted "security" related patches only 22% were true positives, demonstrating low precision and thus, limiting its practicality for downstream fine-grained classification.

GraphSPD identified 403 security patches, of which DUALLM classified 142 as UAF/OOB. Manual verification confirmed that 90 of these are true UAF/OOB patches, yielding an overall precision of **63.4%**. If 31 false positives were excluded due to incorrect GraphSPD classifications (that is, these were not actually security-related as verified manually), we would retain 111 candidates, resulting in a filtered precision (excluding GraphSPD errors) of **81.1%**. Using VulFixMiner as the binary classifier, 110 security patches were identified, and DUALLM reported 39 UAF/OOB cases. Among them, 20 were true positives and 16 were non-security patches, resulting in an overall precision of 51.3%, and a filtered precision of 86.9%.

We also evaluated TreeVul with the same settings for comparison. GraphSPD + TreeVul yielded 104 UAF/OOB predictions, with only 25 true positives and 51 non-security cases, resulting in 24.0% overall precision and 47.2% filtered precision. With VulFixMiner + TreeVul, 51 patches were classified as UAF/OOB, of which 20 were correct and 18 were non-security, yielding a 39.2% overall precision and a 60.6% filtered precision.

Lastly, we include SID for comparison, which does not require a separate binary classifier. SID correctly identified 42 UAF / OOB patches with a precision of 78%, but missed the majority due to its limited coverage of the rules.

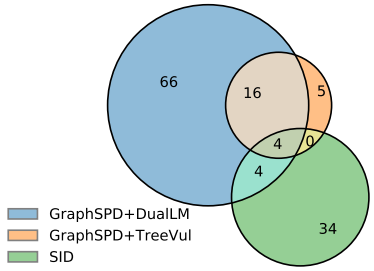

Fig. 13: The Venn diagram for GraphSPD+DualLM, GraphSPD+TreeVul and SID

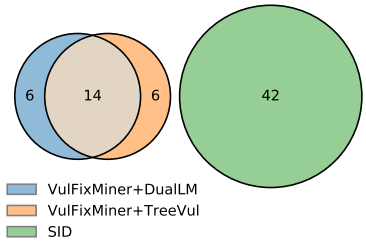

Fig. 14: The Venn diagram for VulFixMiner+DualLM, VulFixMiner+TreeVul and SID

2) True positive study: To better understand the overlap between true positives identified by different pipelines, we constructed Venn diagrams (Figure 13 and Figure 14) based on combinations of binary classifiers (GraphSPD or VulFixMiner) and fine-grained vulnerability classifiers (DUALLM or TreeVul), with SID included for reference. For ease of visual exposition, we split the comparisons by binary classifier. In Figure 13, we show results for GraphSPD-based pipelines and SID. DUALLM identifies 90 true UAF/OOB positives, TreeVul identifies 25, and SID identifies 42. Notably, 64% (20 out of 25) of TreeVul's true positives overlap with DUALLM, showing that most of the true positives identified by TreeVul can also be identified by DUALLM. Manual inspection of TreeVul-only positives reveals that these cases can be reliably classified using only the patch diffs—TreeVul's sole input—while additional contextual information (e.g., slicing) may introduce noise. The limited overlap with SID stems largely from the limited agreement between SID and GraphSPD (only 8 overlapping patches are identified by GraphSPD as security).

Figure 14 presents a similar comparison using VulFixMiner as the binary classifier. Here, DUALLM and TreeVul identify 20 true positives each, with 14 in common, and in this case, SID contributes a fully disjoint set of 42 true positives. The absence of any overlap between SID and VulFixMiner explains the disconnect in the results. Specifically, the disparity arises because VulFixMiner targets Java and Python projects, while SID—like our work—focuses on C/C++ codebases such as the Linux kernel. For the same reason, i.e., VulFixMiner's difficulty with C/C++ codebases, the subsequent results with both DUALLM and TreeVul are more modest compared to that with GraphSPD. In other words, as one might expect, the quality of binary classifiers can significantly affect follow-up results with a fine-grained classifier.

3) False negative study: While the Venn diagrams highlight the coverage and overlap of different pipelines in identifying true positives, it is equally important to understand the limitations of our system—particularly its false negatives. To this end, we further investigate the negative cases from the GraphSPD + DUALLM pipeline to assess whether any true UAF or OOB patches were missed. We randomly sample 100 patches (representing approximately 40% of the total patches labeled as non-UAF-OOB) labeled as non-UAF-OOB and find one false negative (possibly due to a hallucination of the LLM), yielding a **1%** false negative rate.

4) Case study: Interestingly, while 8 of the 90 UAF/OOB patches identified by GraphSPD + DUALLM pipeline, fix bugs that were reported by Syzbot [35] (a continuous fuzzing platform that has reported thousands of Linux kernel bugs), these bugs were not reported as use-after-free or memory out-of-bounds vulnerabilities. Developing reproducers or exploits for kernel vulnerabilities is challenging, particularly since many cases involve race conditions or hardware-specific contexts (such as Dell or NVIDIA devices). In spite of this, we picked two of the syzbot bugs that are not reported as UAF or OOB, and successfully modified the reproducers reported on syzbot to get the PoCs confirming the UAF and OOB behaviors. Most notably, we also developed one exploit against one UAF bug (out of the two cases) which can successfully achieve a **control flow hijacking attack**. This demonstrates the real-world impact of our approach.

Next, we describe how we develop the control flow hijacking attack to exploit the UAF bug. Figure 15 illustrates a

```
netfilter: ipset: Missing gc cancellations fixed

The patch fdb8e12cc2cc ("netfilter: ipset: fix performance
regression in swap operation") missed to add the calls to gc
 cancellations at the error path of create operations and at
 module unload. Also, because the half of the destroy
operations now executed by a function registered by call_rcu
(), neither NFNL_SUBSYS_IPSET mutex or rcu read lock is held
 and therefore the checking of them results false warnings.

 ...
@@ -2378,6 +2379,7 @@ ip_set_net_exit(struct net *net)
       set = ip_set(inst, i);
       if (set) {
          ip_set(inst, i) = NULL;
+         set->variant->cancel_gc(set);
          ip_set_destroy_set(set);
       }
    }
  ...
```

Fig. 15: A use-after-free patch identified by DUALLM

```
1  static int ip_set_create(struct sk_buff *skb, const struct
   nfnl_info *info, ...) {
2   ...
3   cleanup:
4  +    set->variant->cancel_gc(set);
5       set->variant->destroy(set);
6   ...
7  }
8
9  // free `h`
10 static void  mtype_destroy(struct ip_set *set) {
11      struct htype *h = set->data;
12      ...
13      kfree(h);
14      set->data = NULL;
15 }
16
17 static void mtype_cancel_gc(struct ip_set *set) {
18      struct htype *h = set->data;
19      if (SET_WITH_TIMEOUT(set))
20          cancel_delayed_work_sync(&h->gc.dwork);
21 }
22
23 static void expire_timers(struct timer_base *base, struct
   hlist_head *head) {
24      ...
25      struct timer_list *timer;
26      void (*fn)(struct timer_list *);
27      // timer is a dangling pointer
28      timer = hlist_entry(head->first, struct timer_list,
   entry);
29      fn = timer->function;
30      ...
31      // Control flow hijacking
32      call_timer_fn(timer, fn, baseclk);
33 }
```

Fig. 16: Exploitable primitive

patch addressing a bug identified by DUALLM as a use-after-free vulnerability. Note that the commit title/message and code diffs do not directly reveal the bug type, and therefore it was ultimately classified by SLICELM.

As shown in Figure 16, the function call to `mtype_destroy()` on line 5 ultimately frees the `htype` object on line 13. Without `cancel_gc()` being invoked on line 4 —which calls `mtype_cancel_gc()` to terminate the worker process —the worker process would eventually be triggered after a timeout. Because the `htype` object has already been freed, an attacker could exploit this UAF vulnerability by performing heap spraying to gain control of the freed `htype` memory. Specifically, the `timer` pointer, derived on line 28, points to the freed memory. By overwriting `timer->function`, which is a function pointer, an attacker can achieve a control flow hijack on line 32 when it is de-referenced. We developed a working exploit that sprayed `user_key_payload` objects (which are elastic) in kmalloc-2k slabs, to achieve the overwrite and successfully achieved the control flow hijack.

VII. CONCLUSION

In this paper, we presented DUALLM, a Dual-method approach for identifying patches fixing critical memory bugs(e.g., memory out-of-bounds access and use-after-free). Based on our observation that patches can be naturally divided into those with and without clear indicators, DUALLM strategically combines an LLM's capabilities to understand natural language and recognize patch patterns, with a specialized model (SLICELM) trained on our custom slices. Our comprehensive evaluations demonstrate DUALLM's effectiveness. It achieves an 87.4% accuracy on a quality-controlled CVE dataset, significantly outperforming existing approaches. Most importantly, in analyzing 5,140 recent patches, DUALLM identified 90 memory out-of-bounds and use-after-free patches, with two proof-of-concept programs and one exploit achieving successful control flow hijacking, confirming their severity. This underscores the practical impact of our system in identifying crucial patches.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments and valuable suggestions. This work was supported by the National Science Foundation under Grants No. 2155213, the Army Research Office under grant 79845NCH, the Air Force Research Laboratory project under contract FA875025CB041, and the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590041.

REFERENCES

[1] A deep dive into CVE-2023-2163: How we found and fixed an eBPF Linux Kernel Vulnerability. https://bughunters.google.com/blog/6303226026131456/
[2] CVE-2016-5400. https://nvd.nist.gov/vuln/detail/cve-2016-5400.
[3] CVE-2020-27786 (Race Condition + Use-After-Free). https://ii4gsp.github.io/cve-2020-27786/.
[4] CVEs and the NVD Process. https://nvd.nist.gov/general/cve-process#:~:text=Founded%20in%201999%2C%20the%20CVE,Infrastructure%20Security%20Agency%20(CISA).
[5] CWE-119. https://cwe.mitre.org/data/definitions/119.html.
[6] Driving forward in Android drivers. https://googleprojectzero.blogspot.com/2024/06/driving-forward-in-android-drivers.html.
[7] FFmpeg. https://ffmpeg.org/.
[8] Git -diff-config Documentation. https://git-scm.com/docs/diff-config.
[9] GPT-4 Turbo in the OpenAI API. https://help.openai.com/en/articles/8555510-gpt-4-turbo-in-the-openai-api/.
[10] Introducing ChatGPT. https://openai.com/blog/chatgpt.
[11] Introducing LLaMA: A foundational, 65-billion-parameter large language model. https://ai.facebook.com/blog/large-language-model-llama-meta-ai/.
[12] io_uring/kbuf: defer release of mapped buffer rings. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c392cbecd8ec.
[13] Joern Documentation. https://docs.joern.io/home.
[14] Linux Kernel Exploitation. https://github.com/xairy/linux-kernel-exploitation.
[15] Llama 2. https://llama.meta.com/llama2.
[16] Llama 3.1. https://ai.meta.com/blog/meta-llama-3-1/.
[17] OpenAI O1 system card. https://cdn.openai.com/o1-system-card.pdf/.

[18] OpenSSL. https://www.openssl.org/.

[19] SSD Advisory – Linux Kernel taprio OOB. https://ssd-disclosure.com/ssd-advisory-linux-kernel-taprio-oob/.

[20] Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/next/process/submitting-patches.html.

[21] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[22] T. Ahmed and P. Devanbu. Better patching using llm prompting, via self-consistency. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1742–1746. IEEE, 2023.

[23] K. Alrashedy, A. Aljasser, P. Tambwekar, and M. Gombolay. Can llms patch security issues?, 2024.

[24] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[27] W. Chen, X. Zou, G. Li, and Z. Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1093–1110, 2020.

[28] J. Corbet. What to do about CVE numbers. https://lwn.net/Articles/801157/.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[30] T. Dunlap, E. Lin, W. Enck, and B. Reaves. Vfcfinder: Pairing security advisories and patches. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1128–1142, 2024.

[31] S. Feng and C. Chen. Prompting is all your need: Automated android bug replay with large language models, 2023.

[32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[33] M. Fu, V. Nguyen, C. K. Tantithamthavorn, T. Le, and D. Phung. Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types. *IEEE Transactions on Software Engineering*, 2023.

[34] T. Ganz, E. Imgrund, M. Härterich, and K. Rieck. Pavudi: Patch-based vulnerability discovery using machine learning. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 704–717, 2023.

[35] Google. Syzbot. https://syzkaller.appspot.com/upstream/.

[36] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[37] X. He, S. Wang, P. Feng, X. Wang, S. Sun, Q. Li, and K. Sun. Bingo: Identifying security patches in binary code with graph representation learning, 2023.

[38] T. Hoang, J. Lawall, R. J. Oentaryo, Y. Tian, and D. Lo. Patchnet: a tool for deep patch classification. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 83–86. IEEE, 2019.

[39] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*, 2023.

[40] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[41] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[42] H. Li, Y. Hao, Y. Zhai, and Z. Qian. The hitchhiker's guide to program analysis: A journey with large language models, 2023.

[43] H. Li, Y. Hao, Y. Zhai, and Z. Qian. Enhancing Static Analysis For Practical Bug Detection: An LLM-Integrated Approach. In *In Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2024.

[44] X. Li, Z. Zhang, Z. Qian, T. Jaeger, and C. Song. An investigation of patch porting practices of the linux kernel ecosystem. *arXiv preprint arXiv:2402.05212*, 2024.

[45] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[46] T. Linus. Linux Kernel Coding Style. https://slurm.schedmd.com/coding_style.pdf.

[47] T. G. Nguyen, T. Le-Cong, H. J. Kang, R. Widyasari, C. Yang, Z. Zhao, B. Xu, J. Zhou, X. Xia, A. E. Hassan, X.-B. D. Le, and D. Lo. Multi-granularity detector for vulnerability fixes, 2023.

[48] OpenAI. Gpt-4 technical report, 2023.

[49] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li. Fine-grained commit-level vulnerability type prediction by cwe tree structure. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 957–969. IEEE, 2023.

[50] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1–18. IEEE Computer Society, 2022.

[51] C. Qin, A. Zhang, Z. Zhang, J. Chen, M. Yasunaga, and D. Yang. Is chatgpt a general-purpose natural language processing task solver? *arXiv preprint arXiv:2302.06476*, 2023.

[52] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[53] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[54] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. L. Traon. Learning to catch security patches, 2020.

[55] Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang. Lprotector: An llm-driven vulnerability detection system, 2024.

[56] J. Sun, Z. Xing, Q. Lu, X. Xu, L. Zhu, T. Hoang, and D. Zhao. Silent vulnerable dependency alert prediction with vulnerability key aspect explanation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 970–982. IEEE, 2023.

[57] X. Tang, Z. Chen, K. Kim, H. Tian, S. Ezzini, and J. Klein. Just-in-time detection of silent security patches, 2024.

[58] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. ICSE'12.

[59] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks, 2024.

[60] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[61] J. Wang, Z. Huang, H. Liu, N. Yang, and Y. Xiao. Defecthunter: A novel llm-driven boosted-conformer-based code vulnerability detection mechanism, 2023.

[62] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li. Graphspd: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 604–621. IEEE Computer Society, 2022.

[63] X. Wang, K. Sun, A. Batcheller, and S. Jajodia. Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492. IEEE, 2019.

[64] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 149–160. IEEE, 2021.

[65] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*, pages 595–600. IEEE, 2021.

[66] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1914–1927, 2018.

[67] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[68] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.
[69] Y. Wen, J. Cao, and S. Cheng. Ptracer: A linux kernel patch trace bot. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1210–1211. IEEE, 2019.
[70] Q. Wu, Y. He, S. McCamant, and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *NDSS*, 2020.
[71] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 781–797, 2018.
[72] C. S. Xia and L. Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.
[73] D. Xu, K. Chen, M. Lin, C. Lin, and X. Wang. Autopwn: Artifact-assisted heap exploit generation for ctf pwn competitions. *IEEE Transactions on Information Forensics and Security*, 2023.
[74] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.
[75] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang. White-box compiler fuzzing empowered by large language models. corr, abs/2310.15991, 2023b. doi: 10.48550. *arXiv preprint ARXIV.2310.15991*.
[76] C. Yang, Z. Zhao, and L. Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models, 2024.
[77] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.
[78] Z. Zhang, H. Zhang, Z. Qian, and B. Lau. An investigation of the android kernel patch ecosystem. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.
[79] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.
[80] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan. Colefunda: Explainable silent vulnerability fix identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2565–2577. IEEE, 2023.
[81] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 705–716. IEEE, 2021.
[82] Y. Zhou and A. Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 914–919, 2017.
[83] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–27, 2021.
[84] X. Zou, Y. Hao, Z. Zhang, J. Pu, W. Chen, and Z. Qian. Syzbridge: Bridging the gap in exploitability assessment of linux kernel bugs in the linux ecosystem. In *NDSS*, 2024.
[85] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. In *USENIX Security Symposium*, 2022.
[86] F. Zuo, X. Zhang, Y. Song, J. Rhee, and J. Fu. Commit message can help: security patch detection in open source software via transformer. In *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 345–351. IEEE, 2023.

APPENDIX

A. Model and training details.

Our experiments were on a Ubuntu 20.04.5 LTS server with AMD EPYC 7542 processors and 4 Nvidia GeForce RTX 3080 Ti GPUs. The SLICELM model consists of 140 million parameters, including 6 encoder layers and 6 decoder layers, with a model dimension of 768 and 12 heads. We used the Adam optimizer with a linear learning rate decay schedule for optimization, starting the training with a dropout rate of 0.1. The maximum input sequence length is set to 1024 tokens. In addition, we configured ϵ as 1e-6 and β_2 as 0.98 for the Adam optimizer.

B. List of common key phrases

We compiled a list of common key phrases to prepare the dataset for the step to classify memory corruption patches into memory out-of-bounds access, use-after-free and other memory-corruption patches. The commit titles are first converted to lowercase format. The key phrases for memory out-of-bounds access are “fix out of bounds”, “fix out-of-bound”, “fix buffer overflow”, “fix stack overflow”; the key phrases for use-after-free are “fix use after free”, “fix use-after-free”; the key phrase for other memory-corruption patches are “fix uninit-value”, “fix uninitialize”, “fix memory leak”, “fix null pointer dereference”, “fix null dereference”, “fix null pointer reference”, “fix null-ptr deref”, “fix null-ptr-deref”, “fix null pointer access”, “fix null pointer bug”, “fix null deref”, “fix null-deref”, “fix null-ptr-deref”.

C. Data Cleaning of CWE Labels

The bug type labels are mapped from the CWE labels. However, the direct mapping can introduce errors. First, because CWE labels follow a hierarchy, there are CVEs that received CWEs labels at the intermediate level, which does not allow us to identify its true bug type (this is also observed in prior work [49]). For example, 5 CVEs are labeled as `CWE-20: Improper Input Validation`, but in fact, they are actually out-of-bounds bugs. Second, original CWE labels themselves can be incorrect. For example, CVE-2016-5400 [2] is labeled as `CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer`—a category for bugs that ”reads from or writes to a memory location outside the buffer’s intended boundary” [5]. This label suggests an out-of-bounds bug. However, both its commit message and CVE description explicitly indicate that the bug fixed by the patch is memory leak, which is further confirmed by the patch diff showing the addition of memory free operations along an execution path. Overall, we identified and corrected 86 such cases where bug type labels directly mapped from CWE labels were incorrect.