ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching

Yu Nong[1], Haoran Yang[2], Long Cheng[3], Hongxin Hu[1], and Haipeng Cai[1*]

[1] *University at Buffalo*, [2] *Washington State University*, [3] *Clemson University*

[1] {yunong,hongxinh,haipengc}@buffalo.edu, [2] haoran.yang2@wsu.edu, [3] lcheng2@clemson.edu

Abstract

Timely and effective vulnerability patching is essential for cybersecurity defense, for which various approaches have been proposed yet still struggle to generate valid and correct patches for real-world vulnerabilities. In this paper, we leverage the power and merits of pre-trained language language models (LLMs) to enable automated vulnerability patching using no test input/exploit evidence and without model training/fine-tuning. To elicit LLMs to effectively reason about vulnerable code behaviors, which is essential for quality patch generation, we introduce *vulnerability semantics reasoning* and *adaptive prompting* on LLMs and instantiate the methodology as APPATCH, an automated LLM-based patching system. Our evaluation of APPATCH on 97 zero-day vulnerabilities and 20 existing vulnerabilities demonstrates its superior performance to both existing prompting methods and state-of-the-art non-LLM-based techniques (by up to 28.33% in F1 and 182.26% in recall over the best baseline). Through APPATCH, we demonstrate what helps for LLM-based patching and how, as well as discussing what still lacks and why.

1 Introduction

Software vulnerabilities pose a constant and significant threat to the security of modern cyberspace [15, 16] and they are widespread [19, 27, 30–32]. With the rapid growth in the volume and sophistication of cyberattacks, which are enabled by these vulnerabilities, timely and effective vulnerability patching has become increasingly critical [43, 52, 56]. However, manual approaches to patching are costly [26] and struggle to keep pace with the emergence of new vulnerabilities [25]. Thus, automated solutions are needed to help developers patch vulnerabilities before they are exploited [18, 55].

In this context, various approaches to automated patching have been proposed [14, 18, 20, 34, 43, 54], which generally fall in two major categories: those based on code analysis and those based on deep learning (DL). Among other techniques, state-of-the-art (SOTA) code-analysis-based approaches achieve property-based repair based on access range analysis hence enforcing given safety properties [24] or use sanitizers to extract vulnerability-representing constraints for symbolic-execution-based repair [20]. More recently, inductive property inference along with simple templates [55] is employed to generate vulnerability patches. However, these techniques rely on exploits or vulnerability-triggering test inputs, which may not always be available, especially for emerging (e.g., zero-day) vulnerabilities. They often also require that the code to patch be compilable, which further limits their applicability (e.g., they can not be used before the project becomes complete).

DL-based approaches overcome these limitations and have gained more momentum in recent years [14, 18, 52]. These techniques leverage existing vulnerability datasets to train a DL model that learns known vulnerability-fixing patterns, hence the ability to repair a given vulnerability. Despite their promises, they need sizable and quality training data with vulnerability labels, which is not widely and diversely available [9, 39, 41]. As a result, they do not generalize well to unseen code and often stumble on real-world vulnerabilities [52]. While data augmentation helps [12, 38, 40], recent studies show that the augmented models still suffer low accuracy [37, 38].

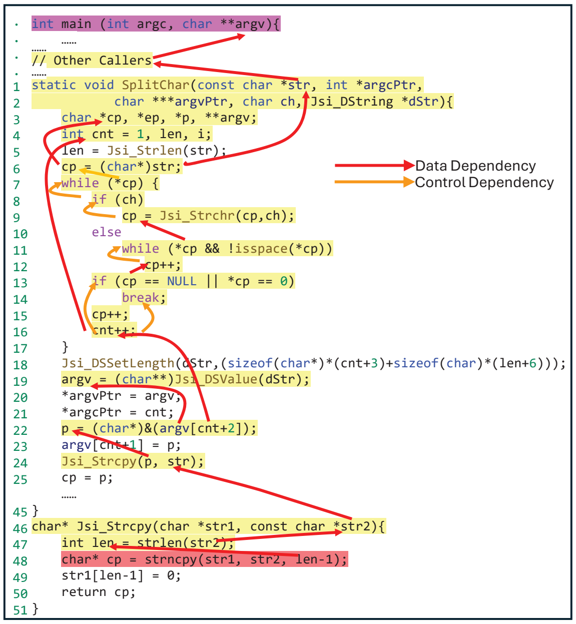
Most recently, large language models (LLMs) have emerged as powerful tools to assist developers with coding related tasks. LLMs overcome the lack of generalizability of ordinary DL-based approaches and have shown promising potential for vulnerability patching [37, 43]. However, the SOTA LLM-based (with zero-shot) technique [43] does not work well on real-world vulnerabilities and often generates no patches or invalid ones [37]. The recent study [37] shows that LLM can be significantly improved by carefully designed prompting strategies, especially based on chain-of-thought (CoT) [51] reasoning. Prompting is indeed attractive in that it does not require (pre-)training or even fine-tuning, which would not only require sizable labeled datasets but also be infeasible to ordinary users, especially on foundation-scale

[*] Haipeng Cai is the corresponding author.

LLMs that may actually hold the promise for effective vulnerability analysis.

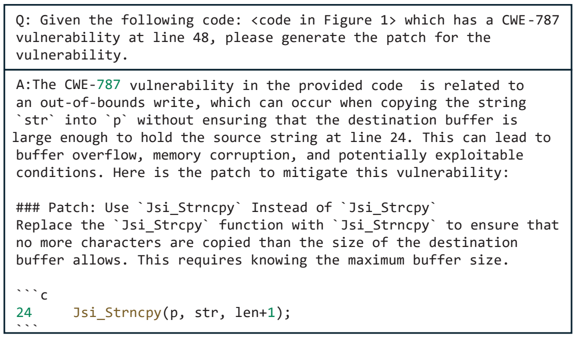
Despite a promising direction, vulnerability patching via prompting LLMs faces several critical challenges as revealed lately [37, 42, 52]. First, for automated patching, the prompting would need to be automated too. Yet it is not known how to automatically write a prompt to an LLM for vulnerability patching (*Challenge 1*). Second, prompting tends to include exemplars to be effective, especially if it aims to elicit reasoning on LLMs [51] as CoT does, which is essential for vulnerability patching—a task relying on code semantics reasoning. Yet again, for an arbitrary program to patch, how to prepare the best exemplars is unknown (*Challenge 2*). Third, LLMs are known to suffer from capacity (#tokens) and context constraints, which makes it difficult to provide necessary code context in a prompt and analyze sizable code; yet vulnerabilities are context-sensitive and can be interprocedural, thus patching them often requires accommodating potentially larger code contexts (*Challenge 3*). Moreover, LLMs are commonly subject to hallucination and non-deterministic responses, whereas stably generating valid and correct patches is essential for vulnerability patching—bad patching can be worse than not patching (*Challenge 4*).

In this paper, we present APPATCH, a novel, automated Adaptive prompting methodology for LLM-based vulnerability patching. Given a vulnerable program with a known vulnerability location, APPATCH first narrows down the scope of analysis to only the relevant subset of the program via a step called *semantics-aware scoping*. This allows LLMs to analyze a smaller code snippet that includes all the essential information necessary for reasoning about the vulnerability-related code behaviors, addressing *Challenge 3*. Then, it elicits the LLM to identify the vulnerability's root cause within the reduced scope with vulnerability semantics reasoning, henceforth selecting exemplars that best fit the program on the fly from a pre-mined exemplar database, an essential step referred to as dynamic *adaptive prompting*, which addresses *Challenges 2*. The database is built offline by mining exemplars from known patches. With the adaptively chosen exemplars, APPATCH forms the patching prompt automatically to generate multiple candidate patches with LLMs, addressing *Challenge 1*. To mitigate model hallucinations, APPATCH consults an ensemble of LLMs to cross-validate the candidate patches concerning multiple facets of patch quality—validity (not changing functionality) and correct (fixing the vulnerability), hence addressing *Challenge 4*.



```
·  int main (int argc, char **argv){
·      ......
·  ......
·  // Other Callers
·  ......
1  static void SplitChar(const char *str, int *argcPtr,
2                char ***argvPtr, char ch, Jsi_DString *dStr){
3      char *cp, *ep, *p, **argv;
4      int cnt = 1, len, i;
5      len = Jsi_Strlen(str);
6      cp = (char*)str;
7      while (*cp) {
8          if (ch)
9              cp = Jsi_Strchr(cp,ch);
10         else
11             while (*cp && !isspace(*cp))
12                 cp++;
13         if (cp == NULL || *cp == 0)
14             break;
15         cp++;
16         cnt++;
17     }
18     Jsi_DSSetLength(dStr,(sizeof(char*)*(cnt+3)+sizeof(char)*(len+6)));
19     argv = (char**)Jsi_DSValue(dStr);
20     *argvPtr = argv;
21     *argcPtr = cnt;
22     p = (char*)&(argv[cnt+2]);
23     argv[cnt+1] = p;
24     Jsi_Strcpy(p, str);
25     cp = p;
       ......
45 }
46 char* Jsi_Strcpy(char *str1, const char *str2){
47     int len = strlen(str2);
48     char* cp = strncpy(str1, str2, len-1);
49     str1[len-1] = 0;
50     return cp;
51 }
```

Figure 1: An example vulnerable program sample where the vulnerable statement is at line 48.



Q: Given the following code: <code in Figure 1> which has a CWE-787 vulnerability at line 48, please generate the patch for the vulnerability.

A:The CWE-787 vulnerability in the provided code is related to an out-of-bounds write, which can occur when copying the string `str` into `p` without ensuring that the destination buffer is large enough to hold the source string at line 24. This can lead to buffer overflow, memory corruption, and potentially exploitable conditions. Here is the patch to mitigate this vulnerability:

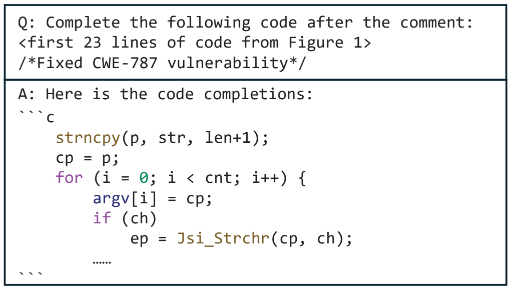
### Patch: Use `Jsi_Strncpy` Instead of `Jsi_Strcpy`
Replace the `Jsi_Strcpy` function with `Jsi_Strncpy` to ensure that no more characters are copied than the size of the destination buffer allows. This requires knowing the maximum buffer size.

```c
24     Jsi_Strncpy(p, str, len+1);
```

Figure 2: GPT-4's patch for the program sample in Figure 1 with *standard prompting*.

We have implemented APPATCH as an open-source tool based on four latest and most powerful LLMs: GPT-4, Gemini-1.5, Claude-3.5, and Llama-3.1, and evaluated it against a dataset with 97 zero-day (across 18 unique projects) samples and a dataset with 20 existing real-world vulnerability samples. With APPATCH, the evaluated LLMs achieve up to 36.46% and 73.86% F1 on the two datasets while the same LLMs with baseline prompting strategies only achieve up to 28.41% and 68.54% F1, respectively. APPATCH also outperforms SOTA traditional vulnerability patching approaches, where they only achieve up to 17.53% and 85.00% recall on the two datasets while APPATCH achieves 49.48% and 90.00% recall, respectively. APPATCH is also reasonably efficient, taking on average 37.148-50.209 seconds to generate a patch, and using 5,684-6,802 tokens per patch with the LLMs.

Our results suggest the practical feasibility and merits of LLM-based vulnerability patching, via a novel methodology (pivoted by *automated adaptive prompting based on vulnerability semantics reasoning*) of leveraging LLMs to generate valid and correct patches. Through APPATCH, we demonstrated how LLMs can be helped for this challenging task; also, by examining failure cases, we discuss how/where and why LLMs may still fall short even with APPATCH.



````
Q: Complete the following code after the comment:
<first 23 lines of code from Figure 1>
/*Fixed CWE-787 vulnerability*/
````

~~~
A: Here is the code completions:
```c
    strncpy(p, str, len+1);
    cp = p;
    for (i = 0; i < cnt; i++) {
        argv[i] = cp;
        if (ch)
            ep = Jsi_Strchr(cp, ch);
        ……
```
~~~

Figure 3: GPT-4's patch for the sample in Figure 1 with a state-of-the-art LLM-based approach *zero-shot completion*.

2 Background and Motivation

LLMs have been explored for vulnerability analysis tasks such as vulnerability detection [36], patching [43, 47], and secure code generation [21]. Among existing approaches to vulnerability patching, a popular method is to use a standard prompting strategy, which directly asks the LLMs to patch the vulnerable code [47]. However, this method does not provide necessary guidance to LLMs, leaving all the vulnerability analysis reasoning process to the models. Another, SOTA method is zero-shot code completion [43], which removes the vulnerable code and lets LLMs complete the vulnerable parts. However, this approach treats vulnerability patching as a simple code completion task, without code analysis/reasoning.

To show the limitations of existing LLM-based vulnerability patching approaches, we test the two methods mentioned above on the example in Figure 1. As shown, the example has a CWE-787 vulnerability at line 48. The vulnerability is caused by the possible out-of-bound write to the pointer `str1` as the string `str2` may be longer than the boundary of (the memory region pointed to by) `str1`. Figure 2 shows GPT-4's output patch with standard prompting, where we directly ask the model to patch the vulnerability with the vulnerable line and CWE ID provided. As seen, GPT-4 seems to recognize that the vulnerability is caused by the insufficient size of the memory *p* points to at line 24 and it should ensure that the write is not longer than `p`'s boundary. While the basic idea is correct, it is also needed to find out how much memory `p` should point to for patching the vulnerability. In this case, necessary data/control dependency analysis is needed. However, GPT-4 generates the patch by changing `Jsi_Strcpy` to `Jsi_Strncpy` and adding a write limit `len+1` without analyzing the code semantics. Thus, it fails to patch the vulnerability.

Figure 3 shows GPT-4's output patch with the zero-shot code completion method. As shown, we provide the code up to line 23 (where the ground-truth patch is located) and prompt the model (via a comment) to complete the remaining code while addressing the CWE-787 vulnerability. Similar to standard prompting, it simply completes line 24 with `strncpy` and sets a write limit `len+1` without analyzing the actual size of `p`. Thus, it also fails to patch the vulnerability.

Based on the two examples above, it is clear that, to patch real-world vulnerabilities effectively, even powerful LLMs may still need to be guided step by step for necessary reasoning about code semantics. Therefore, we introduce APPATCH, an automated framework that adaptively prompts LLMs for vulnerability patching with semantics reasoning guidance. We introduce four key design elements in APPATCH:

1. To help LLMs concentrate on the most essential parts of a program for vulnerability patching, we introduce semantics-aware scoping, which only provides LLMs with code entities that capture the core vulnerable behavior of the program (noted as *vulnerability semantics*).
2. To guide LLMs to analyze and patch vulnerabilities correctly, we arm them with vulnerability semantics reasoning capabilities via CoT prompting [51] that demonstrates correct reasoning steps with a few exemplars.
3. Considering the diversity of vulnerability root causes and patching strategies, a large-scale exemplar pool is needed. However, manually writing the reasoning steps for the exemplars is time-consuming. Thus, we design an automated exemplar mining module which generates exemplars based on existing real-world samples.
4. To provide the best-fit exemplars for a given testing sample (i.e., code to patch), we design dynamic adaptive prompting which automatically picks exemplars based on the vulnerability root causes of the testing sample.

3 Technical Design

Now we present our technical approach, starting with a design overview followed by details on each component.

3.1 Overview

The overall design of APPATCH is depicted in Figure 4. We target a realistic vulnerability patching scenario with APPATCH, in which (1) the vulnerability manifestation locations (i.e., *vulnerable statements*) and (2) the vulnerability types (i.e., *CWE IDs*) of a given program to patch are available. Prior to the patching time, APPATCH demonstrates to LLMs correct reasoning towards patch generation with (3) existing vulnerable samples for which the ground-truth patches are known in addition. These three constitute the **inputs** to APPATCH.

With these inputs, APPATCH operates in two phases. In **Phase 1: Exemplar Mining**, through *semantics-aware scoping* (**1.1**), it union-slices [8] each given vulnerable sample to capture its *vulnerability semantics* (essential behavior of the vulnerability, as formally defined in §3.2). Then, APPATCH proceeds with *exemplar generation* (**1.2**), producing a pool of exemplars each including the root cause analysis, fixing strategy, and ground-truth patch. The root cause and fixing strategy are also generated by the LLMs automatically, according to the vulnerability semantics *along with the ground-truth patch*.

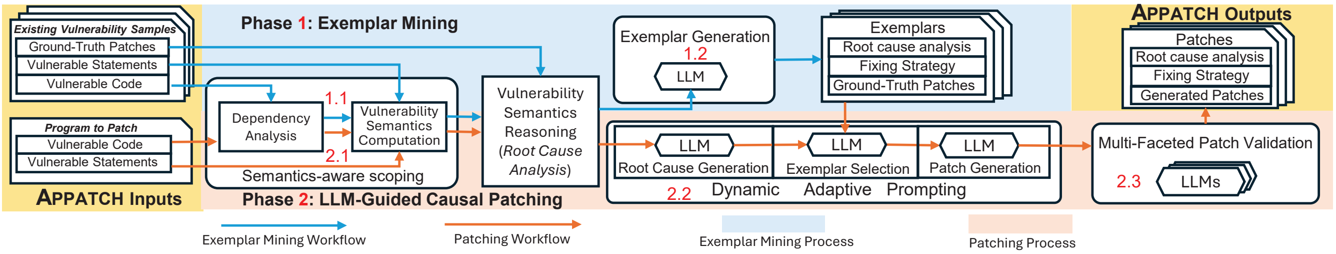


Figure 4: An overview of APPATCH's design, including its inputs, two main working phases, and outputs.

Next, in **Phase 2: LLM-Guided Causal Patching**, the given program to patch is also scoped into vulnerability semantics (**2.1**). Then, through *dynamic adaptive prompting* (**2.2**), APPATCH analyzes the root cause of the given vulnerability with an LLM and selects similar exemplars based on the root cause. With the chosen exemplars, candidate patches are generated via CoT prompting on the LLM. Finally, the generated patches are validated and refined through multiple LLMs through a *multi-faceted patch validation* (**2.3**) process.

As a result, the validated patches are returned as the **outputs** of APPATCH. Note that while both phases rely on LLMs for automated root cause and fixing strategy identification using the vulnerability semantics reasoning, Phase 1 additionally feeds the ground-truth patches to the models, This addition is essential because it is expected to help LLMs generate root cause and fixing strategy that are more likely correct than otherwise hence serving the demonstrating (exemplar) purposes to elicit LLMs' similar reasoning at patching time.

3.2 Vulnerability Semantics

We introduce this core concept underlying APPATCH with following insights. First, although a vulnerable program may contain many lines of code, oftentimes only a relatively small portion of those lines actually causes the vulnerability. By focusing on this portion, which scopes the semantics of the code that makes it vulnerable (intuitively noted as *vulnerability semantics* for now) we can enable LLMs to perform more effective vulnerability analysis. Second, mainstream LLMs often struggle with processing and generating long texts. By narrowing the scope to vulnerability semantics, we can also reduce the length of the exemplars, prompts, and responses, thereby minimizing potential distractions for the LLMs and improving their performance. Third, vulnerability-relevant control/data flow is what is difficult for LLMs to understand without guidance on reasoning about the flow [44]—thus, vulnerability semantics-based reasoning is necessary to guide LLMs to conduct effective vulnerability patching.

Therefore, we define vulnerability semantics as the aspects of a program's code that contribute to its vulnerabilities. These of course include statements where the vulnerabilities are triggered. However, such statements alone do not cause vulnerabilities unless they can be exploited via external inputs. Thus, to effectively capture vulnerability semantics, it is important to consider how these statements are influenced by the program's external inputs through control and data flow dependencies. Therefore, we also include the contextual statements that influence the vulnerability statements as part of vulnerability semantics, based on the data and control flow.

Given a program P, which consists of a set S of statements, the set of **vulnerability locations** $S_v \subseteq S$ are statements where the vulnerability manifests. The set of **external inputs** $EI \subseteq S$ consists of statements where external input data is received or retrieved by P. This includes program input variables, returns from external function calls that read data, and global states modified by other functions (such as memory allocations, file input, network input, etc.). The **vulnerability semantics** $\mathcal{V}(P,S_v,EI)$ is defined as the union of all **static backward slices** of the program P starting from any of the vulnerability locations S_v and terminating at any of the external inputs in EI. That is,

$$\mathcal{V}(P,S_v,EI) = \bigcup_{e_i \in EI} \bigcup_{s_v \in S_v} \text{Slice}(P,s_v,e_i)$$

where:

- $\text{Slice}(P,s_v,e_i)$ is the set of all statements in the static backward slice from s_v to e_i.
- A **static backward slice** from s_v to e_i is the subset of statements in P that affect the execution at s_v, tracing backward along control and data dependencies to reach e_i.

Note that the vulnerability locations and external inputs may be in different functions. Therefore, we perform *interprocedural* backward slicing to capture the vulnerability semantics comprehensively. In this way, $\mathcal{V}(P,S_v,EI)$ represents the set of all statements in P that are relevant to the flow of data and control from any of the external inputs EI to any of the vulnerability locations S_v.

As an example, Figure 1 illustrates a partial representation of vulnerability semantics. We start from a vulnerable statement (s_v) at line 48 and conduct union slicing to the program input based on data and control dependencies (marked with red and orange arrows). As a result, the statements marked as yellow are the contexts of the vulnerable statement. The vulnerable statement (marked in red), the external input (in

Algorithm 1: Semantics-Aware Scoping

```
  Input: $P$: input program, $S_v$: vulnerable statements
  Output: $\mathcal{V}(P, S_v, EI)$: vulnerability semantics (code representation)
1 Function SemanticsAwareScoping($P, S_v, EI$)
2     $sdg \leftarrow$ ExtractSDG($P$);                         // Construct SDG for the program $P$
3     $EI \leftarrow$ IdentifyExternalInputs($P$);               // Identify external inputs
4     $\mathcal{V} \leftarrow$ InitializeSlice();                // Initialize an empty set
5     foreach $e_i$ in $EI$ do
6         foreach $s_v$ in $S_v$ do
7             $BS_{ei} \leftarrow$ ComputeBackwardSlice($sdg, s_v, e_i$);   // Backward slice
                starting from a vulnerable location $s_v$ until reaching an external input $e_i$
8             $\mathcal{V} \leftarrow \mathcal{V} \cup BS_{ei}$;          // Unionize the above backward slice
9     return $\mathcal{V}$;
```

purple), and its contextual statements constitute (the *code representation* of) the vulnerability semantics for the sample.

3.3 Exemplar Mining (Phase 1)

3.3.1 Semantics-Aware Scoping (Step 1.1)

In this step, APPATCH extracts the essential code that represents the vulnerability semantics from each of the existing vulnerable code samples. To that end, it first analyzes the sample and constructs its system dependence graph (SDG) [23] through an *interprocedural dependence analysis*. An SDG is a representation that combines both data dependencies and control dependencies of a program; thus, it is essential in computing the vulnerability semantics of a program. As shown in Algorithm 1, the *vulnerability semantics computation* begins this process by building the SDG (line 2). Then, we perform a simple static analysis to realize `IdentifyExternalInputs` (line 3). We first curate a set of external functions that may produce external inputs or cause side effects to program states (e.g., malloc, socket_recv, scanf). Then, it traverses the program's control flow graph (CFG), identifying each callsite of any of these external functions. To be more comprehensive, it also considers the definition sites of program's input variables as an external input (we simply treat the program entry point as the joint entity of those definition sites).

As the main loop begins, we compute the interprocedural backward slice from each of the vulnerable statements s_v to each of the external input e_i (lines 5-8), via another static analysis based on the SDG just constructed. It traverses the *sdg* from s_v, identifies backward dependencies transitively until e_i is reached (which has no further backward dependence). Finally, we compute the union slice [8] for all external inputs in *EI* (line 8) and return it as the (code representation of the) vulnerability semantics (line 9).

3.3.2 Exemplar Generation (Step 1.2)

With the vulnerability semantics computed in Step 1.1, Step 1.2 generates the exemplars from the given existing vulnerable samples. Since our dynamic adaptive prompting leverages CoT prompting which guides LLMs to generate quality patches with appropriate intermediate reasoning, we divide the overall reasoning process into three high-level steps: (1) *vulnerability semantics reasoning (as root cause analysis)*, (2) fixing strategy identification, and (3) patch generation. However, it is difficult to write the reasoning for each collected sample manually. Therefore, we leverage LLMs to automatically generate the exemplars. Since we have the ground-truth patches in the given existing samples, we prompt the LLMs to generate (1) with the following template:

> **Q**: Given the following code slice $\mathcal{V}_{exem}$, which has a vulnerability among <CWE-IDs> and lines S_v, the patch is <ground-truth patch>. *Starting with the external inputs: <EI identified>, reason about the vulnerable behavior step by step until the vulnerability is determined.*

Since the code slice $\mathcal{V}$ is interprocedural across the whole program, it is likely to contain a large number of code statements [10], making it difficult for LLMs to process. Therefore, the *EI identified* are only part of the full set of EI identified in Algorithm 1 that reach any of the patch locations. Then, the code slice provided to LLMs $\mathcal{V}_{exem}$ only contains the paths between any of such identified EI subset and any of the vulnerable statements. This helps LLMs concentrate on the most essential parts of the program for vulnerability patching.

In this template (as well as the one for dynamic adaptive prompting's root cause generation), we leverage vulnerability-semantics-guided reasoning, which is described as "*Starting with the external inputs, reason about the vulnerable behavior step by step until the vulnerability is determined*". Although this is simple, it significantly helps LLMs analyze the vulnerability root cause comprehensively and effectively. Figures 5 and 9 show the comparison between the root cause analysis with and without vulnerability-semantics-guided reasoning. Without such reasoning, the output misses many details for the vulnerability causes, which make the analysis inaccurate.

The key insight underlying our automated exemplar generation is that, with the ground-truth patches provided, LLMs are capable of generating the correct reasoning steps which are helpful to guide themselves to generate patches given new samples to patch. To corroborate this hypothesis, we conduct a preliminary experiment on the real-world samples from our collected dataset (see §5.1). Based on our manual inspection on the generated exemplars by GPT-4, 92.98% of the reasoning are correct, indicating that using LLMs to automatically generate exemplars is effective. These exemplars compose the quality exemplar pool for the dynamic adaptive prompting step (Step 2.2) to select exemplars from.

3.4 LLM-Guided Causal Patching (Phase 2)

3.4.1 Semantics-Aware Scoping (Step 2.1)

In Phase 2, given a testing sample that contains vulnerable code and its vulnerability location, APPATCH again computes the vulnerability semantics with the semantics-aware scoping module. The process is the same as the one in Step 1.1: we construct the SDG from the source code, then get the vulnerability semantics via Algorithm 1.

3.4.2 Dynamic Adaptive Prompting (Step 2.2)

With the vulnerability semantics computed in Step 2.1, we perform dynamic adaptive prompting to guide the LLMs to generate patches. The overall algorithm of this dynamic adaptive prompting step is shown in Algorithm 2. APPATCH leverages a *progressive prompting* strategy which prompts the LLMs to generate the root cause, select exemplars, and generate the patch in separated prompts. This is to (1) guide the LLMs to generate the patches step by step and (2) allow LLMs to obtain the information and exemplars on demand.

Algorithm 2: Dynamic Adaptive Prompting

```
   Input: vul_slice: vulnerable slice, exemplars: mined exemplars
   Output: cand_patches: candidate patches
1  Function DynPrompting(vul_slice, exemplars)
2      cause ← GenCause(vul_slice)                        // Generate root cause of the slice
3      chosen_exemplars ← []                              // Initialize an empty list
4      foreach exemplar in Exemplars do
5          exemplar_cause ← GetExemCause(exemplar)        // Extract cause
6          comp_ans ← CompareCause(cause, exemplar_cause)
7          if comp_ans is Yes then
8              chosen_exemplars.insert(exemplar)          // Add similar exemplars
9          if SizeOf(chosen_exemplars) ≥8 then
10             break                                      // Break when the exemplars are enough
11     cand_patches←GenPatch(chosen_exemplars, vul_slice)
12     return cand_patches;
13 Function GenCause(vul_slice)
14     cause_slice←GetVulFuncSlice(vul_slice)             // Get vulnerable function slice
15     LLMAns ← GenCauseByLLM(cause_slice)                // Get response from LLMs
16     while HasExpandDemand(LLMAns) do
17         cause_slice ← cause_slice ∪ GetDemandedFuncSlice(LLMAns)
18         LLMAns ← GenCauseByLLM(cause_slice)
19     return LLMAns
```

Root Cause Generation. We again leverage vulnerability semantics reasoning to generate root cause analysis: given the vulnerability semantics computed (i.e., the union slice noted as $\mathcal{V}_{cause}$), the CWE IDs, and the vulnerable statements S_v, APPATCH first prompts the LLMs to generate the root cause (i.e., the natural-language description of the vulnerability semantics reasoning steps). We prompt the LLM with the following template:

> **Q**: Given the following code slice: $\mathcal{V}_{cause}$ which has a vulnerability among <CWE-IDs> and lines: S_v. *Starting with the external inputs: <EI identified>, reason about the vulnerable behavior step by step until the vulnerability is determined.* If you encounter uncertainty due to a lack of function definitions, please tell the functions needed with the format {"context_funcs":[func_1,func_2,CALLER_of_func...]} where "CALLER_of_func" is a placeholder for the caller of the given functions.

In the prompt above, we start from the slice of the function involving the vulnerable statements (line 14 in Algorithm 2) and expand the slice on LLMs' demand. Note that the EI identified are only the part of EI identified in Algorithm I that are relevant to the functions the LLM asked for so far. The rationale is that LLMs usually have input token limitation and thus we cannot input the whole projects into the models. Even if the models are capable of accepting more tokens, the irrelevant functions would distract the LLMs in analyzing the vulnerabilities. Therefore, we only provide the part of the slice that involves the functions that the LLM asks for. This progressive prompting scheme, similar to LLift [28], is essential for APPATCH to achieve *interprocedural* vulnerability analysis and patching.

The algorithm for generating the root cause is shown in Algorithm 2 lines 13-19. Given the vulnerability code slice which may contain a large number of functions and statements, we start with the part of the slice only involving the functions that contain the vulnerability manifestation statements (line 14). Then, we iteratively query LLMs with the prompt template given earlier until the LLMs do not ask for additional functions (lines 15-18). The final root cause analysis is then returned to the main routine (*DynPrompting*) for dynamic adaptive prompting.

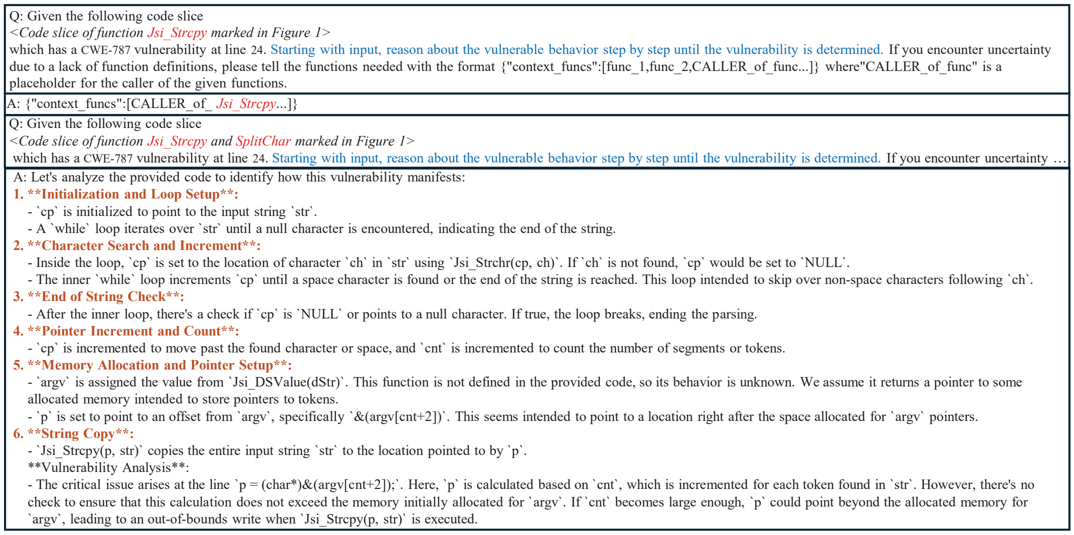
Exemplar Selection. After generating the root cause analysis for the testing sample, we select appropriate exemplars to prompt the LLMs to generate the patches. As shown in Algorithm 2, an empty list, *chosen_exemplars*, is initialized to store exemplars that closely match the root cause (line 3). The selection process iterates through the set of pre-mined exemplars and gets the vulnerability root causes from them (lines 4-5). To select the exemplars, we prompt the LLMs to compare the root cause analysis result of the testing sample and each exemplar sample (line 6) with the following prompt:

> **Q**: Are the following two root causes similar?
> <exemplar_sample_root_cause>
> <testing_sample_root_cause>
> Please simply answer yes or no.

Considering that the complexity is $O(mn)$ where m is the number of exemplar samples and n is that of testing samples, we let LLMs directly answer yes or no to save resources.

Figure 6 shows parts of the exemplar selection process for the example in Figure 1. Once the used LLM answers yes, the root cause analysis, fixing strategy, and the ground-truth patch of that exemplar sample will be used as the exemplar to prompt the LLMs to patch the testing sample (lines 7-8). We follow the original CoT work [51] and only select up to 8 exemplars for each testing sample.

Patch Generation. Once the exemplars are selected, APPATCH prompts the LLMs to generate the patch (line 11) with the following template:



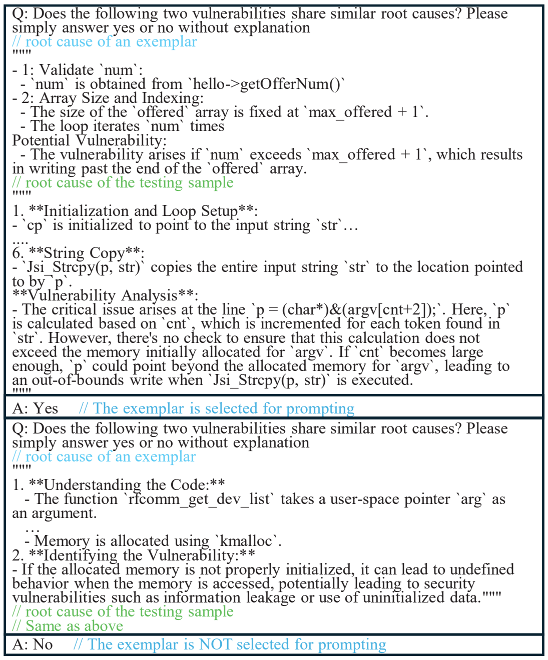
~~~
Q: Given the following code slice
<Code slice of function Jsi_Strcpy marked in Figure 1>
which has a CWE-787 vulnerability at line 24. Starting with input, reason about the vulnerable behavior step by step until the vulnerability is determined. If you encounter uncertainty
due to a lack of function definitions, please tell the functions needed with the format {"context_funcs":[func_1,func_2,CALLER_of_func...]} where"CALLER_of_func" is a
placeholder for the caller of the given functions.
~~~

~~~
A: {"context_funcs":[CALLER_of_ Jsi_Strcpy...]}
~~~

~~~
Q: Given the following code slice
<Code slice of function Jsi_Strcpy and SplitChar marked in Figure 1>
 which has a CWE-787 vulnerability at line 24. Starting with input, reason about the vulnerable behavior step by step until the vulnerability is determined. If you encounter uncertainty …
~~~

~~~
A: Let's analyze the provided code to identify how this vulnerability manifests:
1. **Initialization and Loop Setup**:
   - `cp` is initialized to point to the input string `str`.
   - A `while` loop iterates over `str` until a null character is encountered, indicating the end of the string.
2. **Character Search and Increment**:
   - Inside the loop, `cp` is set to the location of character `ch` in `str` using `Jsi_Strchr(cp, ch)`. If `ch` is not found, `cp` would be set to `NULL`.
   - The inner `while` loop increments `cp` until a space character is found or the end of the string is reached. This loop intended to skip over non-space characters following `ch`.
3. **End of String Check**:
   - After the inner loop, there's a check if `cp` is `NULL` or points to a null character. If true, the loop breaks, ending the parsing.
4. **Pointer Increment and Count**:
   - `cp` is incremented to move past the found character or space, and `cnt` is incremented to count the number of segments or tokens.
5. **Memory Allocation and Pointer Setup**:
   - `argv` is assigned the value from `Jsi_DSValue(dStr)`. This function is not defined in the provided code, so its behavior is unknown. We assume it returns a pointer to some
   allocated memory intended to store pointers to tokens.
   - `p` is set to point to an offset from `argv`, specifically `&(argv[cnt+2])`. This seems intended to point to a location right after the space allocated for `argv` pointers.
6. **String Copy**:
   - `Jsi_Strcpy(p, str)` copies the entire input string `str` to the location pointed to by `p`.
   **Vulnerability Analysis**:
   - The critical issue arises at the line `p = (char*)&(argv[cnt+2]);`. Here, `p` is calculated based on `cnt`, which is incremented for each token found in `str`. However, there's no
   check to ensure that this calculation does not exceed the memory initially allocated for `argv`. If `cnt` becomes large enough, `p` could point beyond the allocated memory for
   `argv`, leading to an out-of-bounds write when `Jsi_Strcpy(p, str)` is executed.
~~~

Figure 5: Root cause analysis generated by GPT-4 ***with*** *vulnerability semantics reasoning* for the sample in Figure 1.



~~~
Q: Does the following two vulnerabilities share similar root causes? Please
simply answer yes or no without explanation
// root cause of an exemplar
"""
- 1: Validate `num`:
  - `num` is obtained from `hello->getOfferNum()`
- 2: Array Size and Indexing:
  - The size of the `offered` array is fixed at `max_offered + 1`.
  - The loop iterates `num` times
Potential Vulnerability:
  - The vulnerability arises if `num` exceeds `max_offered + 1`, which results
in writing past the end of the `offered` array.
// root cause of the testing sample
"""
1. **Initialization and Loop Setup**:
- `cp` is initialized to point to the input string `str`…
....
6. **String Copy**:
- `Jsi_Strcpy(p, str)` copies the entire input string `str` to the location pointed
to by `p`.
**Vulnerability Analysis**:
- The critical issue arises at the line `p = (char*)&(argv[cnt+2]);`. Here, `p`
is calculated based on `cnt`, which is incremented for each token found in
`str`. However, there's no check to ensure that this calculation does not
exceed the memory initially allocated for `argv`. If `cnt` becomes large
enough, `p` could point beyond the allocated memory for `argv`, leading to
an out-of-bounds write when `Jsi_Strcpy(p, str)` is executed.
"""
~~~

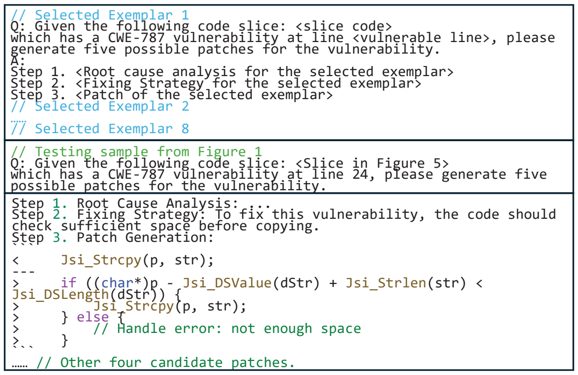
~~~
A: Yes    // The exemplar is selected for prompting
~~~

~~~
Q: Does the following two vulnerabilities share similar root causes? Please
simply answer yes or no without explanation
// root cause of an exemplar
"""
1. **Understanding the Code:**
  - The function `rfcomm_get_dev_list` takes a user-space pointer `arg` as
an argument.
  …
  - Memory is allocated using `kmalloc`.
2. **Identifying the Vulnerability:**
- If the allocated memory is not properly initialized, it can lead to undefined
behavior when the memory is accessed, potentially leading to security
vulnerabilities such as information leakage or use of uninitialized data."""
// root cause of the testing sample
// Same as above
~~~

~~~
A: No    // The exemplar is NOT selected for prompting
~~~

Figure 6: Exemplar selection by GPT-4 for the sample in Figure 1.

<selected exemplars>
Q: Given the following code slice: $\mathcal{V}_{cause}$ which has a vulnerability among <CWE-IDs> and lines: S_v, please generate five possible patches for the vulnerability.
A: Step 1. <root cause analysis>



~~~
// Selected Exemplar 1
Q: Given the following code slice: <slice code>
which has a CWE-787 vulnerability at line <vulnerable line>, please
generate five possible patches for the vulnerability.
A:
Step 1. <Root cause analysis for the selected exemplar>
Step 2. <Fixing Strategy for the selected exemplar>
Step 3. <Patch of the selected exemplar>
// Selected Exemplar 2
……
// Selected Exemplar 8
~~~

~~~
// Testing sample from Figure 1
Q: Given the following code slice: <Slice in Figure 5>
which has a CWE-787 vulnerability at line 24, please generate five
possible patches for the vulnerability.
~~~

~~~
Step 1. Root Cause Analysis: ...
Step 2. Fixing Strategy: To fix this vulnerability, the code should
check sufficient space before copying.
Step 3. Patch Generation:
```
<     Jsi_Strcpy(p, str);
---
>     if ((char*)p - Jsi_DSValue(dStr) + Jsi_Strlen(str) <
Jsi_DSLength(dStr)) {
>         Jsi_Strcpy(p, str);
>     } else {
>         // Handle error: not enough space
>     }
```
…… // Other four candidate patches.
~~~

Figure 7: GPT-4's patch for the sample in Figure 1 with APPATCH.

In this prompt, we first provide the selected exemplars where each exemplar includes its vulnerability semantics (union slice), CWE-IDs, vulnerable statements, as well as the root cause analysis, the fixing strategy, and the ground-truth patch. This is to guide the LLMs to patch the testing sample with a similar, complete workflow. Then, we provide the testing sample's vulnerability semantics slice ($\mathcal{V}_{cause}$) used for the final root cause generation, CWE-IDs, and the vulnerable statements (S_v) with the same format as the exemplars. We also prompt the LLMs to generate more than one candidate patch so that the recall can be improved and developers have more choices to select appropriate patches from. To avoid repeated output, we also *add the root cause generated in the previous step* for the LLMs to follow through. With the dynamically selected exemplars, the LLMs are expected to generate quality patches.

3.4.3 Multi-Faceted Patch Validation (Step 2.3)

After generating the candidate patches, we further leverage multi-faceted (i.e., fixing vulnerability and preserving functionality) patch validation to reduce incorrect patches which are not useful for developers. Considering that the same LLM may not be effective for validating the generated patches by itself, we leverage the ensemble method to conduct the validation. Therefore, multiple effective LLMs for vulnerability patching are used for the validation.

Algorithm 3: Multi-Faceted Patch Validation

```
   Input: cand_patches: candatate patches, llms: LLMs used for validation
   Output: valid_patches: validated patches
1  Function PatchValidation(cand_patches, llms)
2      valid_patches gets cand_patches
3      foreach cand_patch in cand_patches do
4          retain ← False
5          foreach llm in llms do
6              val_ans ← ValPatchLLM(cand_patch, llm) // Validate the patch
                 with an LLM
7              if val_ans is Yes then
8                  retain←True // Remove the patch if all the LLMs think it is invalid
9          if retain is False then
10             valid_patches.remove(cand_patch)
11     return valid_patches
```

Q: Given the following code slice: <vulnerability
semantics slice in Figure 5> which has a CWE-787 vulner-
ability at line 24, please validate whether the following patch fixes the vulnerability while
keeping the functionality: <patch>. Please simply answer yes or no.

```
<    Jsi_Strcpy(p, str);
---
>    if ((char*)p - Jsi_DSValue(dStr) + Jsi_Strlen(str) < Jsi_DSLength(dStr)) {
>        Jsi_Strcpy(p, str);
>    } else {
>        // Handle error: not enough space
>    }
```

A: Yes // By GPT-4	A: yes // By Gemini	A: Yes // By Claude3	A: Yes // By Llama3

Figure 8: Multi-faceted patch validation for the sample in Figure 1.

Algorithm 3 shows the workflow for the validation. It initializes by considering all candidate patches as potentially valid (line 2). It then enters a nested loop where each candidate patch is evaluated independently against each LLM to ascertain its validity (lines 3-4). Within the inner loop, each patch undergoes a validation check by an individual LLM, which determines if the patch is effective (line 5). Each LLM evaluates whether each generated patch fixes the vulnerability correctly with the following prompt:

> Q: Given the following code slice: $\mathcal{V}_{cause}$ which has a vulnerability among <CWE-IDs> and lines: S_v. Please validate whether the following patch fixes the vulnerability while keeping the functionality: <patch>. Please simply answer yes or no.

In the prompt, we also ask the LLMs whether the patch keeps the functionality to avoid valid but incorrect patches.

Figure 8 shows the multi-faceted patch validation prompts and answers for the example in Figure 1. If any LLM approves the patch, a flag (retain) is set to True, indicating that the patch should be retained in the list of valid patches (lines 6-8). If after evaluation by all the LLMs, a patch is not validated by any (i.e., the retain flag remains False), it is removed from the list of valid patches (lines 9-10). This step ensures that only those patches that have been validated by at least one LLM are kept. The algorithm concludes by returning the refined list of validated patches, which have passed through this rigorous validation process (line 11). We use such an approach rather than majority voting because APPATCH aims to improve the recall while keeping the precision, which is more practically valuable than otherwise for real-world developers. Majority voting may improve the precision but reduce the recall, which is less helpful for real-world patching.

Table 1: LLMs used in our study

Model	Version	#Params	Max #Tokens	Vendor	Release date	Cutoff Date
Gemini-1.5	gemini-1.5-pro	50T	30,720	Google	May 2024	Nov. 2023
Claude-3.5	claude-3.5-sonnet	70B	200,000	Anthropic	Jun. 2024	Apr. 2024
GPT-4	gpt-4-turbo	175B+	128,000	OpenAI	May 2024	Oct. 2023
Llama-3.1	llama-3.1-70b	70B	4,096	Meta	July 2024	Dec. 2023

4 Implementation

To support the generation of *interprocedural* vulnerability semantics, we use Joern [53] to construct the SDG of the given vulnerable program under patching. In this work, since we evaluate APPATCH on a number of C code samples without compiling them, we chose Joern which is a powerful tool designed for scalable code analysis without requiring compilability of the input code. Specifically, we first obtain the PDG (which includes the control and data dependencies) for each function (which Joern immediately provides). Then, we leverage the call graph also provided by Joern to conduct the interprocedural dependence analysis hence building the SDG. With the SDG, we extract the interprocedural backward slice from the vulnerable statements. In this work, we reuse the script from SySeVR [33] to quickly query the control flow nodes for the respective statements and find the in-degree edges and nodes. Based on this, we further implement the semantics-aware scoping algorithm, as shown in Algorithm 1. The extracted vulnerability semantics slices are stored as sliced source code text for LLMs to use.

5 Evaluation

We evaluate APPATCH via four research questions (RQs):

- **RQ1:** How effective is APPATCH for real-world vulnerabilities patching?
- **RQ2:** How does each of the key design components of APPATCH contribute to its overall performance?
- **RQ3:** How effective is APPATCH compared to existing vulnerability patching techniques?
- **RQ4:** How efficient is APPATCH for patch generation?

We first describe the LLMs, datasets and metrics used. Then, we answer the four research questions above.

LLMs. Table 1 summarizes the four SOTA LLMs chosen for our study to cover the latest models of a variety of sizes and vendors. **Gemini-1.5** [45] and **Claude-3.5** [5] are the latest LLMs from Google and Anthropic, respectively. We chose their most popular powerful (`Pro` and `Sonnet`) for both effectiveness and efficiency. **GPT-4** is the latest and most powerful LLM from OpenAI and we use its most powerful version `GPT-4-Turbo`. **Llama3.1** [46] is an SOTA LLM from Meta, of which we also chose to use the powerful version llama-3.1-70b for our experiments. We chose general-purpose LLMs instead of code-specific ones (e.g., CodeLlama) because the latter are less capable of code analysis. As shown in Appendix Table 9, our preliminary experiments revealed that APPATCH, when based on popular code-specific LLMs (CodeLlama [2], CodeQwen 1.5 [3], and DeepSeek-Coder-v2 [57]), achieved only 1.21%, 10.12%, and 9.09% F1 scores, respectively, significantly lower than the performance of general-purpose LLMs when used in APPATCH as base LLMs as shown in Table 2.

Datasets. To collect exemplar samples for Phase 1 (*Exemplar Mining*), we use existing widely used vulnerability datasets **PatchDB** [49] and **CVEFixes** [9], where PatchDB contains 12073 real-world fixing samples and CVEFixes contains 4,120 real-world fixing samples in C language. In this work, we select the samples with the most popular CWEs in C languages: CWE-787 (out-of-bound write), CWE-125 (out-of-bound read), CWE-190 (integer overflow), CWE-401 (memory leak), CWE-457 (use of uninitialized variable), and CWE-476 (use of NULL pointer). However, some of the fixes contain edits not relevant to vulnerabilities (e.g., code refactoring or adding new functionalities). Therefore, we manually inspect each of the samples and filter out the inappropriate samples. To be realistic, we read the vulnerability reports and label vulnerability manifestation locations, rather than the code changed lines in the vulnerability fixing commits. We applied an inter-rater agreement/consensus procedure—each author independently labeled them, followed by cross-checking outcomes and discussions to resolve disagreements. We also remove the overlapped samples between PatchDB and CVEFixes, as well as confirming that there is no overlap between the training set and the testing sets discussed below. As a result, we collect 306 vulnerability fixing samples.

For the testing set, considering that the latest powerful LLMs are trained on existing data, it is possible that they have seen the samples in PatchDB and CVEFixes during the training. Thus, there may be data leakage and contamination issues if we use the samples from PatchDB and CVEFixes as the testing set. Therefore, we collect a **Zero-Day dataset** where all the vulnerabilities were reported after the latest cutoff date of the LLMs (04/2024 from Claude-3.5-Sonnet). As a result, we collect 97 vulnerabilities which also cover the same CWEs as PatchDB and CVEFixes. Among them, 21 vulnerabilities are interprocedural where the vulnerability manifestation locations and the patching locations are in different functions. These samples cover 18 open-source projects including Linux Kernal, FFmpeg, etc. Appendix Table 8 shows the detailed statistics of the dataset.

To compare APPATCH with existing vulnerability patching techniques, we also use an existing vulnerability dataset **ExtractFix** [20] for comparison experiments. The samples from ExtractFix are compilable and come with test cases that allow us to validate patches. From this dataset, we collect 20 vulnerabilities that can be reproduced, for each of which we label the manifestation locations and the CWE IDs based on the crash information from the test cases. Detailed statistics of this dataset can be also found in Appendix Table 8. We also addressed data leakage concerns with it in Appendix E.

Metrics. Since the dataset covers many projects and it is difficult to compile each of the samples and use the test cases to validate the generated patches (furthermore, some of the samples do not come with test cases), we cannot automatically validate the generated patches if they do not exactly match the ground truths. Therefore, we manually check each generated patch. To improve the manual evaluation and better understand the generated patches, we consider four metrics: (1) if the generated patch exactly matches the ground truth, we consider it as *syntactic equivalent (SynEq)*; (2) if the generated patch does not exactly match the ground truth but the behavior is the same as the ground truth, we consider it as *semantically equivalent (SemEq)*; (3) if the patch has different behavior from the ground truth but still fix the vulnerability without breaking the code functionality, we consider it as *plausible*; (4) all the samples with the three metrics above are considered as *correct*. To better simulate the real-world patching scenarios, we prompt the LLMs to generate up to five patches so that developers can pick up the one most suitable to patch the vulnerable code. To better evaluate the generated patches, we consider three measures:

$$recall = \frac{\#fixed\ samples}{\#testing\ sample} \tag{1}$$

$$precision = \frac{\#correct\ patches}{\#generated\ patches} \tag{2}$$

$$F1 = \frac{2 \times recall \times precision}{recall + precision} \tag{3}$$

where *fixed sample* means a sample where at least one of the generated patches fix the vulnerability.

5.1 RQ1: Effectiveness

Table 2 shows the effectiveness of APPATCH against the two testing datasets. We notice that APPATCH achieves the best overall effectiveness for vulnerability patching. With APPATCH, GPT-4, Gemini-1.5, Claude-3.5, and Llama-3.1 achieve 33.30%, 20.44%, 36.46%, and 24.28% F1 for the correctness metric on our collected Zero-Day dataset, which is the best among other prompting approaches. On the existing ExtractFix dataset, APPATCH again achieves the best among other prompting approaches, with 68.41%, 42.90%, 73.86%, and 64.20% F1 for the correctness metric, respectively. The overall effectiveness on ExtractFix dataset is higher, indicating the challenges on Zero-Day vulnerability patching.

Table 2: Effectiveness of APPATCH

Model	Approach	Zero-Day Dataset												ExtractFix Dataset											
		SynEq			SemEq			Plausible			Correct			SynEq			SemEq			Plausible			Correct		
		Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1
GPT-4	APPATCH	6.19%	1.64%	**2.59%**	26.80%	14.48%	**18.80%**	25.77%	12.84%	**17.14%**	39.18%	28.96%	**33.30%**	5.00%	2.30%	**3.15%**	75.00%	42.53%	**54.28%**	40.00%	10.34%	**16.44%**	90.00%	55.17%	**68.41%**
	No Validation	7.22%	1.46%	2.42%	28.87%	13.10%	18.02%	28.87%	11.02%	15.95%	44.33%	25.57%	32.43%	5.00%	1.98%	2.84%	80.00%	41.58%	54.72%	45.00%	9.90%	16.23%	90.00%	53.47%	67.08%
	No Slicing	3.09%	0.61%	1.02%	28.87%	8.98%	13.70%	21.65%	9.80%	13.49%	39.18%	19.39%	25.94%	10.00%	1.98%	3.31%	65.00%	41.58%	50.72%	30.00%	11.88%	17.02%	80.00%	55.45%	65.50%
	Random Exemplars	5.15%	1.02%	1.71%	31.96%	11.48%	16.89%	23.71%	6.97%	10.77%	44.33%	19.47%	27.05%	10.00%	1.98%	3.31%	75.00%	35.64%	48.32%	50.00%	17.82%	26.28%	80.00%	55.45%	65.50%
	Manual Exemplars	3.09%	1.03%	1.54%	28.87%	11.11%	16.05%	17.53%	5.97%	8.90%	42.27%	18.11%	25.35%	5.00%	0.99%	1.65%	70.00%	39.60%	50.59%	35.00%	11.88%	17.74%	85.00%	52.48%	64.89%
	Direct Reasoning	3.09%	0.62%	1.03%	22.68%	8.28%	12.13%	19.59%	6.83%	10.13%	36.08%	15.73%	21.91%	5.00%	0.99%	1.65%	75.00%	43.56%	55.11%	30.00%	6.93%	11.26%	80.00%	51.49%	62.65%
	Standard Prompting	3.09%	0.62%	1.03%	18.56%	7.19%	10.36%	17.53%	6.78%	9.77%	30.93%	14.58%	19.82%	5.00%	0.99%	1.65%	65.00%	40.59%	49.98%	30.00%	9.90%	14.89%	65.00%	51.49%	57.46%
	Zero-shot Completion	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	12.37%	6.78%	8.76%	12.37%	6.78%	8.76%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	15.00%	3.96%	6.27%	15.00%	3.96%	6.27%
Gemini-1.5	APPATCH	2.06%	0.56%	**0.88%**	9.28%	6.16%	**7.41%**	21.64%	9.80%	**13.49%**	26.80%	16.53%	**20.44%**	0.00%	0.00%	**0.00%**	35.00%	18.68%	**24.20%**	30.00%	16.48%	**21.28%**	50.00%	35.16%	**42.90%**
	No Validation	2.06%	0.55%	0.87%	9.28%	6.32%	7.52%	21.64%	9.61%	13.31%	26.80%	16.48%	20.41%	0.00%	0.00%	0.00%	35.00%	17.46%	23.30%	30.00%	15.87%	20.76%	50.00%	33.33%	40.00%
	No Slicing	0.00%	0.00%	0.00%	10.31%	5.08%	6.81%	18.56%	10.56%	13.69%	24.74%	15.93%	19.38%	0.00%	0.00%	0.00%	20.00%	15.94%	17.74%	35.00%	18.84%	24.50%	40.00%	34.78%	37.21%
	Random Exemplars	1.03%	0.34%	0.51%	12.37%	5.82%	7.92%	15.46%	8.56%	11.02%	24.74%	14.72%	18.46%	5.00%	1.09%	1.79%	35.00%	17.39%	23.24%	25.00%	10.87%	15.15%	45.00%	29.35%	35.53%
	Manual Exemplars	2.06%	0.68%	1.03%	9.28%	4.11%	5.70%	17.52%	10.96%	13.48%	22.68%	15.75%	18.59%	0.00%	0.00%	0.00%	25.00%	13.79%	17.78%	30.00%	14.94%	19.95%	45.00%	28.74%	35.07%
	Direct Reasoning	1.03%	0.29%	0.46%	8.25%	3.24%	4.66%	20.61%	8.85%	12.39%	25.77%	12.38%	16.73%	0.00%	0.00%	0.00%	45.00%	30.21%	36.15%	10.00%	3.13%	4.76%	45.00%	33.33%	38.30%
	Standard Prompting	1.03%	0.26%	0.41%	11.34%	4.10%	6.03%	18.56%	6.92%	10.08%	25.77%	11.28%	15.69%	0.00%	0.00%	0.00%	20.00%	12.87%	15.66%	30.00%	12.87%	18.01%	45.00%	25.74%	32.75%
	Zero-shot Completion	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.09%	1.21%	1.74%	3.09%	1.21%	1.74%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	20.00%	8.24%	11.67%	20.00%	8.24%	11.67%
Claude-3.5	APPATCH	4.12%	0.92%	**1.51%**	37.11%	17.32%	**23.62%**	22.68%	10.62%	**14.47%**	49.48%	28.87%	**36.46%**	10.00%	2.04%	**3.39%**	80.00%	44.90%	**57.52%**	45.00%	18.37%	**26.09%**	85.00%	65.31%	**73.86%**
	No Validation	5.15%	1.07%	1.78%	37.11%	16.52%	22.87%	22.68%	10.09%	13.96%	50.52%	27.68%	35.77%	10.00%	1.98%	3.31%	80.00%	44.55%	57.23%	45.00%	17.82%	25.53%	85.00%	64.36%	73.25%
	No Slicing	3.09%	0.63%	1.05%	27.84%	12.47%	17.23%	27.84%	10.57%	15.32%	41.24%	23.68%	30.08%	10.00%	1.98%	3.31%	65.00%	44.55%	52.87%	30.00%	7.92%	12.53%	80.00%	54.46%	64.80%
	Random Exemplars	1.03%	0.21%	0.34%	28.87%	15.11%	19.84%	17.53%	6.83%	9.83%	41.24%	22.15%	28.82%	5.00%	0.99%	1.65%	75.00%	43.56%	55.11%	45.00%	16.83%	24.50%	90.00%	61.39%	72.99%
	Manual Exemplars	2.06%	0.42%	0.69%	29.90%	13.13%	18.24%	21.65%	8.13%	11.82%	41.24%	21.67%	28.41%	5.00%	0.99%	1.65%	80.00%	46.53%	58.84%	40.00%	9.90%	15.87%	85.00%	57.43%	68.54%
	Direct Reasoning	2.06%	0.49%	0.80%	23.71%	14.32%	17.86%	17.53%	7.65%	10.66%	37.11%	22.47%	27.99%	5.00%	0.99%	1.65%	75.00%	52.48%	61.75%	35.00%	15.84%	21.81%	75.00%	69.31%	72.04%
	Standard Prompting	2.06%	0.43%	0.72%	24.74%	12.61%	16.70%	14.43%	6.09%	8.56%	28.87%	19.13%	23.01%	15.00%	2.97%	4.96%	55.00%	40.59%	46.71%	25.00%	12.87%	16.99%	65.00%	56.44%	60.42%
	Zero-shot Completion	0.00%	0.00%	0.00%	4.12%	1.12%	1.76%	15.46%	8.28%	10.78%	18.56%	9.40%	12.48%	0.00%	0.00%	0.00%	5.00%	3.49%	4.11%	30.00%	17.44%	22.06%	30.00%	20.93%	24.66%
Llama-3.1	APPATCH	1.03%	0.27%	**0.42%**	21.65%	9.55%	**13.25%**	18.56%	8.75%	**11.89%**	35.05%	18.57%	**24.28%**	5.00%	1.03%	**1.71%**	70.00%	44.33%	**54.28%**	20.00%	8.25%	**11.68%**	80.00%	53.61%	**64.20%**
	No Validation	1.03%	0.22%	0.36%	22.68%	9.41%	13.30%	20.62%	8.09%	11.62%	38.14%	17.72%	24.20%	5.00%	0.99%	1.65%	70.00%	42.57%	52.95%	20.00%	7.92%	11.35%	80.00%	51.49%	62.65%
	No Slicing	2.06%	0.49%	0.79%	20.62%	9.09%	12.62%	11.34%	4.67%	6.61%	28.87%	14.25%	19.08%	0.00%	0.00%	0.00%	70.00%	29.70%	41.71%	30.00%	10.89%	15.98%	75.00%	40.59%	52.68%
	Random Exemplars	1.03%	0.22%	0.37%	15.46%	6.52%	9.17%	9.28%	4.49%	6.06%	23.71%	11.24%	15.25%	5.00%	0.99%	1.65%	65.00%	39.60%	49.22%	10.00%	2.97%	4.58%	75.00%	43.56%	55.11%
	Manual Exemplars	3.09%	0.74%	1.19%	19.59%	11.30%	14.33%	11.34%	4.67%	6.61%	29.90%	16.71%	21.44%	5.00%	0.99%	1.65%	70.00%	34.65%	46.36%	30.00%	11.88%	17.02%	85.00%	47.52%	60.96%
	Direct Reasoning	4.12%	0.83%	1.39%	15.46%	7.71%	10.29%	17.52%	6.45%	9.44%	31.95%	15.00%	20.41%	0.00%	0.00%	0.00%	70.00%	48.51%	57.31%	30.00%	5.94%	9.92%	70.00%	54.46%	61.26%
	Standard Prompting	3.09%	0.63%	1.05%	14.43%	7.34%	9.73%	14.43%	5.87%	8.35%	24.74%	13.84%	17.75%	0.00%	0.00%	0.00%	60.00%	40.59%	48.43%	15.00%	5.94%	8.51%	65.00%	46.53%	54.24%
	Zero-shot Completion	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.28%	4.17%	5.75%	9.28%	4.17%	5.75%	0.00%	0.00%	0.00%	5.00%	1.98%	2.84%	5.00%	2.97%	3.73%	5.00%	4.95%	4.98%

In terms of recall, Claude-3.5 achieves the best with 49.48% on the Zero-Day dataset and GPT-4 acheives the best with 90.00% on the ExtractFix dataset. GPT-4 achieves the best precision 28.96% on the Zero-Day dataset and Claude-3.5 achieves the best precision 65.31% on the ExtractFix dataset. In terms of different metrics, Claude-3.5 has 37.11% and 80.00% recall on semantics equivalent against the Zero-Day and ExtractFix datasets, respectively. This indicates that APPATCH generates patches close to developers' patches.

> APPATCH is effective for vulnerability patching, with up to 36.46% and 73.86% F1 on the Zero-Day and ExtractFix dataset, respectively, showing its practical potential.

Table 3: Effectiveness of APPATCH against the interprocedural samples from the Zero-Day dataset

Approach	Model	Correct			Model	Correct		
		Recall	Prec	F1		Recall	Prec	F1
APPATCH	GPT-4	38.10%	26.67%	**31.37%**	Claude-3.5	47.62%	25.93%	**33.57%**
No Validation		38.10%	25.23%	30.35%		47.62%	25.23%	32.98%
No Slicing		33.33%	18.18%	23.53%		38.10%	21.78%	27.72%
Random Exemplars		33.33%	15.32%	20.99%		47.62%	19.82%	27.99%
Manual Exemplars		38.10%	17.27%	23.77%		42.86%	18.92%	26.25%
Direct Reasoning		38.10%	13.51%	19.95%		42.86%	19.81%	27.10%
Standard Prompting		19.05%	6.60%	9.81%		14.29%	6.86%	9.27%
Zero-shot Completion		9.52%	4.72%	6.31%		14.29%	5.94%	8.39%
APPATCH	Gemini-1.5	28.57%	14.14%	**18.91%**	Llama-3.1	38.10%	17.65%	**24.12%**
No Validation		28.57%	13.34%	18.18%		38.10%	14.41%	20.91%
No Slicing		23.81%	12.37%	16.28%		14.29%	23.08%	17.65%
Random Exemplars		23.81%	11.46%	15.47%		23.81%	10.89%	14.95%
Manual Exemplars		23.81%	11.11%	15.15%		19.05%	15.38%	17.02%
Direct Reasoning		23.81%	7.62%	11.54%		28.57%	10.81%	15.68%
Standard Prompting		4.76%	1.10%	1.79%		9.52%	3.13%	4.71%
Zero-shot Completion		0.00%	0.00%	0.00%		9.52%	2.70%	4.21%

5.2 RQ2: Contributions of Components

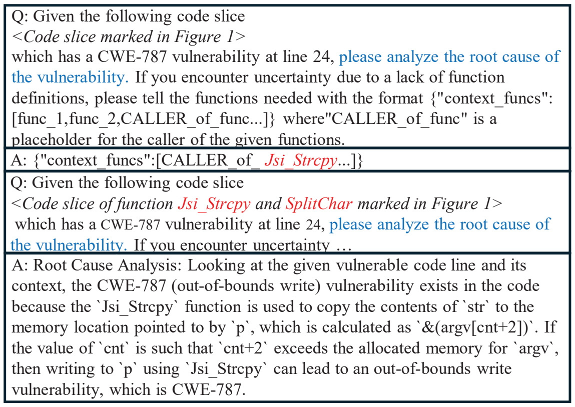
As shown in Table 2, we then examine the contributions of the APPATCH components with the ablated prompting approaches. The numbers marked in red are the best F1 scores achieved by the ablated prompting approaches. One of the ablated versions, "Manual Exemplars", corresponds to our prior work [37], which is *also an LLM-based patching baseline*.

When we remove the multi-faceted patch validation, as shown in the "no validation", we notice that the precision drops and thus the F1 becomes lower, with 20.41%–35.77% F1 on the Zero-Day dataset and 40.00%–73.25% F1 on the ExtractFix dataset, indicating that the multi-faceted patch validation is useful to improve the precision.

We then examine the contribution of semantics-aware scoping. In this ablation study, instead of using the vulnerability semantics slice, we directly input the whole source code sample into the LLMs for analysis. As shown in rows "No slicing", without the vulnerability semantics slicing, the recall, precision, and F1 drop significantly compared with APPATCH, with 19.08%–30.08% F1 on the Zero-Day dataset and 37.21%–65.50% F1 on the ExtractFix dataset. The reason is that, without slicing, the code not relevant to vulnerabilities is also inputted into the LLMs, thus distracting the analysis and impact the accuracy of the root cause analysis. To prove this, we examine the correct rates of the reasoning on the testing samples. As Table 10 shows, vulnerability semantics reasoning achieves higher accuracy if we have vulnerability semantics scoping/slicing. This indicates the importance of inputting core vulnerability semantics for analysis.

To examine the usefulness of dynamic adaptive prompting, we conduct experiments with randomly selected exemplars rather than selecting exemplars based on the root causes of the vulnerabilities. Note that we still keep the CWEs of the exemplars the same as the testing sample. As shown in the rows "Random Exemplars", we notice that the F1 scores drop dramatically compared with APPATCH.The exemplars selected for prompting have promising fixing strategies which help LLMs patch the vulnerability correctly, while they may pick up incorrect fixing strategies to patch with random exemplars. This indicates the importance and effectiveness of dynamic prompting for vulnerability patching.

As the exemplars in our exemplar pool are generated by LLMs, to examine whether these LLM-generated exemplars are better than human curated exemplars, we also manually write exemplars following the same reasoning steps. As shown in rows "Manual Exemplars", the overall effectiveness drops significantly, indicating the effectiveness of adaptive prompting for our LLM-based vulnerability patching design.



Q: Given the following code slice
<Code slice marked in Figure 1>
which has a CWE-787 vulnerability at line 24, please analyze the root cause of the vulnerability. If you encounter uncertainty due to a lack of function definitions, please tell the functions needed with the format {"context_funcs": [func_1,func_2,CALLER_of_func...]} where"CALLER_of_func" is a placeholder for the caller of the given functions.

A: {"context_funcs":[CALLER_of_ *Jsi_Strcpy*...]}

Q: Given the following code slice
<Code slice of function Jsi_Strcpy and SplitChar marked in Figure 1>
which has a CWE-787 vulnerability at line 24, please analyze the root cause of the vulnerability. If you encounter uncertainty …

A: Root Cause Analysis: Looking at the given vulnerable code line and its context, the CWE-787 (out-of-bounds write) vulnerability exists in the code because the `Jsi_Strcpy` function is used to copy the contents of `str` to the memory location pointed to by `p`, which is calculated as `&(argv[cnt+2])`. If the value of `cnt` is such that `cnt+2` exceeds the allocated memory for `argv`, then writing to `p` using `Jsi_Strcpy` can lead to an out-of-bounds write vulnerability, which is CWE-787.

Figure 9: Root cause analysis generated by GPT-4 ***without*** *vulnerability semantics reasoning* for the sample in Figure 1.

To show the effectiveness of vulnerability semantics reasoning, we simply prompt the LLMs to generate root causes, as shown in Figure 9. As shown in the row "Direct Reasoning", the overall effectiveness drops significantly compared to APPATCH. The reason is that the accuracy of root cause analysis is lower than vulnerability semantics reasoning, as shown in Table 10. This indicates that guiding LLMs for vulnerability semantics reasoning is effective for improving vulnerability analysis and patching.

To show the patching effectiveness of the LLMs without any specific prompting design, we directly ask the LLMs to patch the given vulnerabilities. As shown in row "Standard Prompting", we notice that the standard prompting have much lower effectiveness compared with APPATCH, achieving only up to 23.01% F1 on the Zero-Day dataset and 60.42% F1 on the ExtractFix dataset even with the state-of-the-art LLMs.

Recently, Pearce et al. [43] approaches the LLM-based vulnerability repair problem through zero-shot code completion, which removes the vulnerable code in the program and lets the LLMs fill the vulnerable parts so as to fix the vulnerability. We use their best template `s2` for comparison. As shown in the Row "zero-shot completion", this approach has the lowest effectiveness, with up to 12.48% F1 on our Zero-Day dataset and 24.66% F1 on the ExtractFix dataset, indicating that only providing prior code without further information is insufficient for LLMs to fix the vulnerabilities.

> Each component in APPATCH substantially contributes to its effectiveness, justifying its overall design.

We further evaluate APPATCH's effectiveness on interprocedural vulnerabilities using the 21 interprocedural samples from the Zero-Day dataset. Table 3 shows the correctness results (the complete results are in Appendix Table 12). Similarly, the numbers marked in red are the best F1 scores achieved by the ablated prompting approaches. Compared to the overall results on the complete Zero-Day dataset, APPATCH and its ablated versions show a modest decrease in performance (e.g., APPATCH's F1 score with GPT-4 decreases from 33.30% to 31.37%).

Table 4: Comparison to existing techniques in terms of recall

	ExtractFix Dataset				Zero-Day Dataset			
Technique	SynEq	SemEq	Plausible	Correct	SynEq	SemEq	Plausible	Correct
APPATCH	5%	75%	40%	**90%**	4.12%	37.11%	22.68%	**49.48%**
VulRepair	10%	25%	0%	35%	2.06%	15.46%	1.03%	17.53%
Getafix	20%	5%	0%	25%	5.15%	0.00%	1.03%	6.19%
ExtractFix	20%	35%	30%	85%	-	-	-	-
VulnFix	20%	10%	15%	45%	-	-	-	-

In contrast, the baseline approaches—Standard Prompting and Zero-Shot Completion—exhibit a substantial decline in effectiveness (e.g., Claude-3.5 with standard prompting and zero-shot completion decrease from 23.01% and 12.48% to 9.81% and 6.31%, respectively). Note that for Standard Prompting, we provide all the functions covered by the computed vulnerability semantics. For zero-shot completion, we directly provide functions with the ground-truth patch location as it can only work in such a setting. These already benefit the two baselines but they still fall short with these interprocedural samples. This deterioration can be attributed to the absence of interprocedural analysis in these approaches which adversely affects the LLMs' performance.

> While APPATCH has a slight performance drop on exclusively interprocedural vulnerabilities, baseline approaches show a dramatic decline. This highlights the effectiveness merits of interprocedural analysis in APPATCH.

5.3 RQ3: Comparison to Existing Techniques

To compare the effectiveness of APPATCH to existing vulnerability patching techniques, we select four state-of-the-art non-LLM-based tools which cover different technical categories, as shown in Table 4, where the red numbers indicate the best recall achieved by the baselines. VulRepair [18] is a DL-based vulnerability repair technique which fine-tunes a pre-trained programming language model CodeT5 [50] to repair vulnerabilities. Getafix [7] is a data-driven pattern-based vulnerability repair technique which learns vulnerability patching patterns from existing samples. ExtractFix [20] is a traditional vulnerability patching technique which leverages test cases and constraints to fix the vulnerabilities. VulnFix [55] is a SOTA vulnerability patching technique based on inductive inference.

As ExtractFix and VulnFix require compilable projects and test cases which are difficult to build on our Zero-Day dataset, we first use the ExtractFix dataset for comparison. As ExtractFix and VulnFix generate many patches for testing, we only compare the recall metric in this RQ. For VulRepair and Getafix which are learning-based, we use our PatchDB+CVEFixes training set to re-train their models.

As shown in Table 4, on the ExtractFix dataset, APPATCH with the best LLM (GPT-4) achieves the best recall (90%)

compared with the baseline techniques. Note that the recall sum of syntactic equivalent, semantically equivalent, and plausible may be over recall on correct because there are multiple patches generated for each testing sample. ExtractFix achieves 85% recall because it leverages the test cases and constraints for patching. Without such information, other baseline techniques only achieve 25%-45% recall.

Note that the ExtractFix dataset is released in 2021, which is possible to cause the data leakage issue as it is before the cutoff date of the used LLMs. Thus, we test the LLMs whether they know the related information and patches given the meta data. The results show that they either do not know or give wrong answers, indicating that the data leakage issue is not serious. Detailed results can be found in Appendix E.

We further evaluate the baselines on the Zero-Day dataset. Since these samples are not compilable, we limit our comparison to VulRepair and Getafix as the other two (VulnFix and ExtractFix) both require that the input code is compilable. As shown in Table 4, APPATCH significantly outperforms both baselines, achieving a recall of 49.48% compared to VulRepair's 17.53% and Getafix's 6.19%. This highlights APPATCH's superior effectiveness in patching.

> Compared to traditional techniques, APPATCH is effective in vulnerability patching without extra information.

5.4 RQ4: Efficiency

We evaluate the efficiency of APPATCH in terms of time cost and token usage. Table 5 shows the time cost and token usage of APPATCH per sample on each LLM. After mining the exemplars, each sample takes between 37.148 seconds and 50.209 seconds to generate a patch, including the time for selecting the exemplars and generating the candidate patches. In this process, the input/context token usage ranges from 5,684 to 6,802 per sample, and the generated token usage ranges from 584 to 886. We notice that Claude-3.5 and GPT-4 take more tokens while Gemini-1.5 and Llama-3.1 take less. The reason is that the former two models tend to generate more tokens when generating the exemplars and patches. Based on the rate of each model's API, patching each sample costs between 0.2 and 8 cents. However, Gemini-1.5 provides a free version if the usage is not large and Llama-3.1 supports deploying locally. If so, the two models are free.

> APPATCH is efficient for vulnerability patching in terms of both time and money cost.

6 Discussion

We further discuss the advantages of APPATCH that enable its greater effectiveness, as well as why and how it fails on some samples, its tool usability, its current limitations, and the extensibility of APPATCH.

Table 5: Time and token cost of APPATCH per sample

Model	Time cost	Contexts Tokens	Generated Tokens	Price
GPT-4	40.169s	6290	647	$0.0823
Claude-3.5	50.209s	6802	886	$0.0336
Gemini-1.5	37.148s	5684	584	$0.0026
Llama-3.1	46.598s	5749	591	$0.0203

6.1 Why APPATCH Works

There are several advantages of APPATCH for vulnerability patching. The first advantage is contributed by the semantics-aware scoping. As Figure 1 shows, without semantics-aware scoping, the code to be analyzed and patched includes many statements not relevant to the vulnerabilities. In this case, the LLMs are distracted and thus are more difficult to generate correct root cause and patches. In comparison, resulting from our semantics-aware scoping, the slice analyzed and patched by LLMs only includes the core code fragment for vulnerability analysis and patching. This not only helps LLMs concentrate on the vulnerability semantics which is the most important parts for patching, but also makes the data and control flow clearer than the original code. For example, in the original code, the variable `cnt` at line 22 is 6 lines away from its last definition at line 16. However, in the vulnerability semantics slice, it is only two lines away, which makes LLMs easier to find the data dependencies for vulnerability patching.

Figures 2, 3 and 7 show the generated analysis and patch by GPT-4 with standard prompting, zero-shot completion and APPATCH, where the standard prompting and zero-shot completion fail to patch the vulnerability correctly while APPATCH succeeds. We notice that in APPATCH with semantics-aware scoping, GPT-4 is able to capture more data flow information for variables `p`, `cnt`, and `dStr`, while the GPT-4 with standard prompting and zero-shot code completion only analyze and patch the code based on the vulnerable line itself. This indicates that semantics-aware scoping helps LLMs better work on the data dependency analysis, which is important for effective vulnerability patching.

The second advantage of APPATCH is that it prompts dynamically based on the testing samples rather than using random or fixed prompts. This allows LLMs to leverage a large amount of existing real-world vulnerability data with few-shot learning, while avoiding LLM fine-tuning which incurs high costs of time and money. Because of the token limitation and the disadvantage of LLMs for long-text process, it is not feasible to input all the exemplars for few-shot learning. In contrast, the dynamic adaptive prompting with CoT guides LLMs to analyze the vulnerabilities in depth rather than analyzing them superficially. As Figures 2, 3 and 7 show, GPT-4 with APPATCH conducts in-depth analysis for the root cause of the vulnerability, while GPT-4 with standard prompting

Table 6: Accuracy of CodeQL detection, localization, classification, and the three combined against the Zero-Day dataset

Scenario	Detection	Localization	Classification	Combined
Fully Automated	87.62%	59.79%	69.07%	55.67%
Realistic	87.62%	72.16%	83.50%	68.04%

Table 7: End-to-end results against the Zero-Day dataset

Scenario	Model	Approach	Correct		
			Recall	Prec	F1
Fully Automated	Claude-3.5	APPATCH	34.02%	18.36%	***23.85%***
		Manual Exemplars	22.68%	11.33%	*15.11%*
		Standard Prompting	16.49%	11.15%	13.30%
		Zero-shot Completion	15.46%	9.04%	11.41%
	VulRepair	-	16.49%	9.18%	11.79%
	Getafix	-	6.19%	5.70%	5.93%
Realistic	Claude-3.5	APPATCH	41.24%	20.40%	***27.29%***
		Manual Exemplars	25.77%	11.38%	*15.79%*
		Standard Prompting	24.74%	16.16%	19.55%
		Zero-shot Completion	15.46%	9.04%	11.41%
	VulRepair	-	16.49%	9.18%	11.79%
	Getafix	-	6.19%	5.70%	5.93%

and zero-shot completion cannot. The exemplars from dynamic adaptive prompting guide GPT-4 to find the size and boundary of the pointer `p`, which significantly helps generate the patch correctly.

Notably, the ability of LLMs to automatically generate the vulnerability semantics reasoning steps (i.e., the natural language representation of vulnerability semantics) is key to the automation in Step 1.2. Also, since the reasoning essentially dissects the vulnerable behavior of the code, they immediately serve as root cause analysis. Importantly, using *functionality semantics* instead would help the LLMs understand the functional behaviors of the code, hence not helping much the models with understanding the vulnerability root cause hence guiding their patch generation, as we extensively validated.

The third advantage is that APPATCH conducts multi-faceted patch validation strategy to remove the unusable patches. As shown in Table 2, with multi-facted patch validation, the precision is improved significantly without impacting the recall. This improves the F1 and effectively reduces the load of developers for selecting appropriate patches, thus shows the practicability of APPATCH.

6.2 Usability of APPATCH

In this work, we assume that the vulnerabilities under patching are pre-detected with correct CWE and location information. However, such information may not be always available. In this case, APPATCH may need to collaborate with other vulnerability analyzers (that provide the information) to work in practice. Thus, to demonstrate that our formulation of APPATCH is *realistic* and showcase its usability, we evaluate its performance when fed with such information produced from upstream detection/localization and classification tools, a setting referred to as *end-to-end* integration, against our Zero-Day dataset on the best-performing LLM Claude-3.5.

We consider two integration scenarios for a comprehensive evaluation. The first is a fully automated workflow in which both information from the upstream tools is directly fed to APPATCH as its inputs without human intervention. The second is a scenario in which the outputs of those tools are first inspected, verified, and calibrated if necessary, and the validated/calibrated location and CWE information is then fed to APPATCH. In a typical, realistic vulnerability management workflow, developers would inspect and calibrate the outputs of automated (vulnerability localization and classification) tools before proceeding with patching using those outputs, for many reasons (e.g., accuracy/trust issues—since no automated tools are perfect and they can produce false positives and suffer false negatives, among other considerations [4]). Thus, we refer to the second as the realistic scenario.

For the upstream tooling, we chose CodeQL [1] as it is an industry-grade security analyzer widely used on real-world projects. Also, as it serves detection/localization and classification purposes at the same time, users need just one tool instead of multiple to provide both inputs to APPATCH. For CodeQL to work specifically for a given project, users often customize the query packs based on the project information [1]. We also followed this common practice for the projects evaluated. Notably, our query-pack customization did not assume any knowledge about the ground-truth results (i.e., detection/localization/classification outcomes), avoiding biases and keeping the realism of the end-to-end integration setting. We extended the memory manipulation function set in the query packs so that the project's custom memory-management APIs (e.g., `kmalloc`, `kfree`) can be scanned as well—which any ordinary user would be able to do as well for a given project to scan. For the realistic scenario involving developer inspection, we had a graduate student with 3-year relevant experience simulate the realistic human-intervention step between vulnerability localization/classification and patching, who inspected and calibrated CodeQL outputs.

APPATCH's end-to-end performance results in both scenarios are in Table 7 (with further details in Appendix Table 13). To contextualize these results, Table 6 gives the upstream tool performance, which is largely consistent with what earlier studies found on CodeQL [13]. Note that CodeQL may report multiple locations and CWEs for one sample, similar to other vulnerability localization techniques [17, 22, 35]. Therefore, we follow their evaluation approach and report top-10 accuracy for localization and classification where the ranking is based on the security severity [6]. As shown, APPATCH worked reasonably well in these end-to-end settings, with expected accuracy drops versus when perfect vulnerability location and CWE information is used (Table 2). These gaps can be justified by inaccuracies of the upstream results with-/without human inspection. APPATCH also outperforms the baseline approaches, demonstrating its merits and superior practical performance. With more advanced upstream tools to provide more accurate vulnerability locations and CWEs, and users with more expertise (e.g., developers of the project under patching), these performance could be even better.

6.3 How and Why APPATCH Fails

We conduct case studies on the all the failure cases of APPATCH on the Zero-Day dataset to investigate the symptoms of them. There are several major symptom groups.

- **Incorrect Vulnerability Identification.** The most common symptom is incorrect vulnerability identification, with 35.69% of the symptoms. This category includes cases where the initial analysis misidentified the type or nature of the vulnerability. For example, as shown in Figure 10 the actual issue is related to a missing condition check for the debug event code type, not an out-of-bounds write as initially assumed.
- **Insufficient or incorrect code modification.** Another common issue was insufficient or incorrect code modification (31.23% of symptoms). For example, as shown in Figure 11, the patch focuses on local buffer handling while the actual fix require dynamic memory allocation and more comprehensive input validation.

These symptoms show *how* APPATCH fails to patch some samples. We further identified three main root causes:

- **Misunderstanding the actual vulnerabilities.** The most common root cause for failed patching is that the LLMs misunderstand the actual vulnerabilities, with 41.33% of the root causes. This usually causes incorrect vulnerability identification, as shown in Figure 10.
- **Inadequate analysis of code context and dependencies.** Another common root cause for failed patching is inadequate analysis of code context and dependencies (25.21% of root causes). This usually causes insufficient or incorrect code modification, as shown in Figure 11.
- **Failure to consider all edge cases or scenarios.** This category covers situations where the patch didn't account for all possible scenarios or edge cases, with 10.21% of the root causes. For example, in one sample, the LLM incorrectly handles special characters in shell quoting and fails to address all cases of characters requiring special treatment, particularly those needing quoting rather than escaping.

6.4 Limitations

There are several factors that may limit APPATCH in practice. The first limitation is that the root cause analysis may not be accurate at the testing phase. If the vulnerability root cause analysis is incorrect, the direction of patching is misled and the generated patches are less likely to be correct. The second limitation is that we only collect 306+76+20 patching samples for evaluation, because collecting these samples require manual works to label. Yet we still cannot ensure these samples are compilable and come with test cases.

For the first limitation, APPATCH mitigates it by using the state-of-the-art and most powerful LLMs to generate the patches. We exclude the LLMs that are not capable of conducting logical analysis, such as Falcon [58], StarCoder [29], and GPT-NeoX [11]. All the selected LLMs in this work have the capability of analyzing the code logically. For the second limitation, we collected as many as samples as we can so that they can cover more CWEs, vulnerable code patterns, and patching strategies/patterns.

Indeed, the two limitations can be attributed to the vulnerability datasets we use. In an ideal vulnerability patching scenario, the exploits should be provided and the dynamic tracing of the vulnerability should be available. However, the used datasets, PatchDB and CVEFixes, only provide the raw information of the vulnerabilities (e.g., versions, patches, source code) without further information. Considering that reproducing the vulnerabilities based on the CVE reports is very time-consuming, it is difficult for us to build a dataset with hundreds of compilable and vulnerability-reproducable code samples in a short time. At the same time, existing vulnerability datasets coming with compilable code and test cases are relatively small, such as the ExtractFix dataset [20] which only has 30 samples. Therefore, future works should try to build larger compilable vulnerability datasets so that rigorous static and dynamic analysis can be feasible.

6.5 Extensibility

A common concern for the practicability of APPATCH is whether it can be extended to other CWEs and languages. While the current implementation and the evaluation are based on C language with six common CWEs, the design of APPATCH is language/CWE agnostic and is extendable to other CWEs and languages. To extend APPATCH to other CWEs, users can add exemplar samples of other CWEs to generate the respective exemplars. The samples with more CWEs can be also found in the PatchDB and CVEFixes datasets. To extend APPATCH to other languages, users can set up a code analyzer that is able to construct SDGs and then follow Algorithm 1 to extract the vulnerability semantics slices. Meanwhile, our used code analyzer, Joern, also supports more than ten languages, including the most popular languages such as Java, Python, C#, and Go.

6.6 LLMs for Automated Patching

Beyond the merits of APPATCH itself, our work also reveals several insights for future automated patching with LLMs.

The first is that prompting design matters for effective vulnerability patching. Based on our evaluation, simply asking LLMs to patch vulnerabilities work poorly because of the lack of guidance. While vulnerability patching by the SOTA approach [43] shows its potential when LLMs started surging, it performs poorly even with the latest and most powerful LLMs.

This indicates that the prompts for vulnerability patching need to be specifically designed with comprehensive guidance.

The second insight is that LLM-based vulnerability patching still needs traditional code analysis to complement. The main reason is that LLMs usually have token limitation so that we cannot input a large amount of code into the model. Even if some of the models support more tokens, the long input and output text may also distract the model from concentrating the important parts for patching because of the limitation of their base structure-Transformer [48]. Therefore, extracting the important code slices by traditional code analysis techniques is promising for LLM-based vulnerability patching.

The third insight is that dynamically prompting LLMs adaptively is necessary for few-shot learning-based vulnerability patching. The reason is that, because of the token limitation of LLMs, it is difficult to input all the samples covering different vulnerability root causes and patching strategies. The exemplars used for few-shot learning should be based on testing samples to achieve better effectiveness.

Meanwhile, we choose to use LLMs as they are, without pre-training and fine-tuning. The main reason is that the cost of LLMs also matters. If we choose to pre-train our own LLMs, the training data, hardware usage, and time cost would be considerable. Only big companies such as OpenAI and Google have enough resource to pre-train LLMs. Currently, most of the powerful LLMs are commercial, thus it is difficult to access these LLMs directly for fine-tuning. Although some provide APIs for fine-tuning, the models are usually old (e.g., GPT-3.5) and the price is expensive considering the amount of data for fine-tuning. Therefore, prompting LLMs would be more feasible than pre-training and fine-tuning.

7 Related Work

Recent research has focused on example-based approaches to vulnerability detection and repair, demonstrating the effectiveness of learning from past vulnerabilities and their corresponding fixes. Ma et al. [34] presented VuRLE, a system that utilizes machine learning to automatically detect and repair vulnerabilities by learning from examples. Similarly, Zhang et al. [54] developed a similar approach for Java.

Deep learning techniques have also shown significant promise in automating the vulnerability detection and repair process. Chen et al. [14] explored neural transfer learning for repairing vulnerabilities in C code. VulRepair, a T5-based automated software vulnerability repair tool, is presented by Fu et al. [18]. This approach shows significant improvements in the accuracy and efficiency of vulnerability repair.

Program analysis and language models offer powerful techniques for understanding and fixing software vulnerabilities by analyzing code semantics and employing complicated models. Gao et al. [20] proposed a novel method that goes beyond conventional testing by extracting crash constraints to guide program vulnerability repair. Lastly, Pearce et al. [43] examined the application of LLM for zero-shot vulnerability repair. Their study highlights the potential of LLMs to fix vulnerabilities without extensive task-specific training, leveraging the models' ability to generalize from a vast amount of data.

8 Conclusion

We propose APPATCH, an automated vulnerability patching framework that features dynamic adaptive prompting on LLMs to elicit their effective reasoning for vulnerability root-cause analysis hence quality patch generation. We have demonstrated, on four latest LLMs, that APPATCH substantially outperforms both existing prompting approaches and state-of-the-art non-LLM-based patching techniques.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their effective guidance and constructive comments. For this work, Yu Nong, Haoran Yang, and Haipeng Cai were supported by Army Research Office (ARO) under Grant No. W911NF-21-1-002, National Science Foundation (NSF) under Grant No. CCF-2146233 and CCF-2505223, and Office of Naval Research (ONR) under Grant No. N000142212111; Long Cheng was supported by NSF under Grant No. 2239605 and 2228616; and Hongxin Hu was supported by NSF under Grant No. 2228617, 2120369, and 2129164.

Ethics Considerations

Our research is focused on the development and evaluation of methods for patching software vulnerabilities, which complies with the ethical guidelines outlined by the USENIX Security'25 conference. Specifically, we have ensured that our study adheres to principles concerning the responsible handling of security research.

Our study focuses on generating and testing patches for vulnerabilities in software systems. While we evaluate our technique mainly in a zero-day setting (i.e., using the vulnerabilities that are zero-days to the LLMs which have a cutoff date before the disclosure time of these vulnerabilities), the vulnerabilities have already been disclosed (as CVEs) publicly by the time of our paper writing. Thus, our research itself does not cause harm or risks related to vulnerability disclosure, nor raising any new security threats. Nevertheless, throughout our research process, we have taken care to avoid any actions that could result in unintended harm, such as exposing systems or users to new security risks. We have ensured that all vulnerability data utilized is anonymized and does not contain any identifiable information about individuals or organizations. Besides, we have not conducted any experiments on production systems that could impact real-world users, hence avoiding any potential disruption or harm.

Open Science

We fully support the new open science policy of USENIX Security'25, which requires research results to be publicly accessible or for researchers to provide a valid explanation if this is not possible. To align with this policy, we commit to openly sharing all artifacts generated from our research, including the codebase, datasets, and experimental results, through a publicly accessible repository. The source code and documentation of APPATCH as well as our experimental results have been made available at https://zenodo.org/records/14741018.

In summary, our work aligns with both the ethical guidelines and open science policy of USENIX Security'25. We are committed to conducting research that not only adheres to high ethical standards but also promotes openness and community engagement in advancing cybersecurity research.

References

[1] CodeQL documentation. https://codeql.github.com/docs/, 2021.

[2] Introducing Code Llama, A state-of-the-art large language model for coding. https://ai.meta.com/blog/code-llama-large-language-model-coding/, 2023.

[3] Code with CodeQwen1.5. https://qwenlm.github.io/blog/codeqwen1.5/, 2024.

[4] A fool with a tool is still a fool. https://dwheeler.com/flawfinder/, 2024.

[5] Meet claude. https://www.anthropic.com/claude, 2024.

[6] Metadata for CodeQL queries. https://codeql.github.com/docs/writing-codeql-queries/metadata-for-codeql-queries/, 2024.

[7] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *OOPSLA*, 3:1–27, 2019.

[8] Árpád Beszédes, Csaba Faragó, Z Mihaly Szabo, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *ICSM*, pages 12–21, 2002.

[9] Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In *PROMISE*, pages 30–39, 2021.

[10] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *TOSEM*, 16(2):8–es, 2007.

[11] Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of BigScience Episode# 5–Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136, 2022.

[12] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. Generating vulnerable code via learning-based program transformations. In *AI Embedded Assurance for Cyber Systems*, pages 123–138. Springer, 2023.

[13] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. An empirical study of static analysis tools for secure code review. In *ISSTA*, pages 691–703, 2024.

[14] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in C code. *TSE*, 49(1):147–165, 2022.

[15] Ericsson. Software vulnerability: Impact & ways to avoid it. https://www.ericsson.com/en/security/vulnerability-management, 2023.

[16] Forbes Technology Council. Zero-day vulnerabilities: 17 consequences and complications. https://www.forbes.com/councils/forbestechcouncil/2023/05/26/zero-day-vulnerabilities-17-consequences-and-complications, 2023.

[17] Michael Fu and Chakkrit Tantithamthavorn. LineVul: A Transformer-based line-level vulnerability prediction. In *MSR*, pages 608–620, 2022.

[18] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. VulRepair: A T5-based automated software vulnerability repair. In *ESEC/FSE*, pages 935–947, 2022.

[19] Xiaoqin Fu and Haipeng Cai. FlowDist: Multi-staged refinement-based dynamic information flow analysis for distributed software systems. In *USENIX Security Symposium*, pages 2093–2110, 2021.

[20] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *TOSEM*, 30(2):1–27, 2021.

[21] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *CCS*, pages 1865–1879, 2023.

[22] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. LineVD: Statement-level vulnerability detection using graph neural networks. In *MSR*, pages 596–607, 2022.

[23] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, 1990.

[24] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. Using safety properties to generate vulnerability patches. In *S&P*, pages 539–554, 2019.

[25] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. The secret life of software vulnerabilities: A large-scale empirical study. *TSE*, 49(1):44–63, 2022.

[26] Ilan Peleg. The high cost of security vulnerabilities. https://www.forbes.com/sites/forbesbusinesscouncil/2023/04/10/the-high-cost-of-security-vulnerabilities-why-observability-is-the-solution, 2023.

[27] Information Technology Laboratory at NIST. National vulnerability database dashboard. https://nvd.nist.gov/general/nvd-dashboard, 2023.

[28] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An LLM-integrated approach. *OOPSLA*, 8:474–499, 2024.

[29] Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, LI Jia, Jenny Chim, Qian Liu, et al. StarCoder: May the source be with you! *TMLR*.

[30] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. PolyCruise: A cross-language dynamic information flow analysis. In *USENIX Security Symposium*, pages 2513–2530, 2022.

[31] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. PolyFuzz: Holistic greybox fuzzing of multi-language systems. In *USENIX Security Symposium*, pages 1379–1396, 2023.

[32] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. PyRTFuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing. In *CCS*, pages 1645–1659, 2023.

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A framework for using deep learning to detect software vulnerabilities. *TDSC*, 19(4):2244–2258, 2021.

[34] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In *ESORICS*, pages 229–246, 2017.

[35] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. VulChecker: Graph-based vulnerability localization in source code. In *USENIX Security*, pages 6557–6574, 2023.

[36] David Noever. Can large language models find and fix vulnerable software? *arXiv preprint arXiv:2308.10345*, 2023.

[37] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024.

[38] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. VGX: Large-scale sample generation for boosting learning-based software vulnerability analyses. In *ICSE*, pages 1–13, 2024.

[39] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. Generating realistic vulnerabilities via neural code editing: An empirical study. In *ESEC/FSE*, pages 1097–1109, 2022.

[40] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. VulGen: Realistic vulnerable sample generation via pattern mining and deep learning. In *ICSE*, pages 2527–2539, 2023.

[41] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. Open science in software engineering: A study on deep learning-based vulnerability detection. *TSE*, 49(4):1983–2005, 2022.

[42] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. Automated software vulnerability patching using large language models. *arXiv preprint arXiv:2408.13597*, 2024.

[43] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *S&P*, pages 2339–2356, 2023.

[44] Moumita Das Purba, Arpita Ghosh, Benjamin J Radford, and Bill Chu. Software vulnerability detection using large language models. In *ISSREW*, pages 112–119, 2023.

[45] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[46] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[47] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *S&P*, pages 199–199, 2024.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NeurIPS*, 30, 2017.

[49] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. PatchDB: A large-scale security patch dataset. In *DSN*, pages 149–160, 2021.

[50] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, pages 8696–8708, 2021.

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 35:24824–24837, 2022.

[52] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In *ISSTA*, pages 1282–1294, 2023.

[53] Fabian Yamaguchi. A platform for robust analysis of C/C++ code. https://joern.readthedocs.io/en/latest/installation.html, 2022.

[54] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng Yao, and Na Meng. Example-based vulnerability detection and repair in Java code. In *ICPC*, pages 190–201, 2022.

[55] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. Program vulnerability repair via inductive inference. In *ISSTA*, pages 691–702, 2022.

[56] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and roadmap. *TOSEM*, 2024.

[57] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong

Ma, et al. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

[58] Yoshua X ZXhang, Yann M Haxo, and Ying X Mat. Falcon LLM: A new frontier in natural language processing. *AC Investment Research Journal*, 220(44), 2023.

Appendices

A Compostion and Statistics of the Datasets

Table 8 shows the composition and statistics of the two datasets. The Zero-Day dataset covers 18 projects and six CWEs where the average number of patched lines is 7.87. The ExtractFix dataset covers seven projects with the same CWEs, while the average number of patched lines is 3.05.

B Code LLMs against the Zero-Day Dataset

Table 9 shows APPATCH effectiveness on code LLMs against the Zero-Day dataset. For each of the models, we use its most powerful version (with most #paramters)–CodeLlama-70b, CodeQwen-1.5-7b, and DeepSeek-Coder-V2-236b. As shown, the three popular used code LLMs only achieve up to 10.12% F1 score, much lower than the general-purpose LLMs.

C Correct Reasoning Rate Comparison

Table 10 compares the performance of four language models (GPT-4, Gemini-1.5, Claude-3.5, and Llama-3.1) on the Zero-Day dataset using three different prompting approaches. Across all models, APPATCH's *Vulnerability Semantics Reasoning* approach (highlighted in green) consistently achieves the highest correct reasoning rates. The *Direct Reasoning* and *No Slicing* approaches generally yielded lower success rates, demonstrating the effectiveness of the Vulnerability Semantics Reasoning method.

D More Results on Interprocedural Samples

Table 12 shows the detailed results of APPATCH against the interprocedural samples from the Zero-Day dataest, with red numbers indicating the best F1 scores achieved by the ablated prompting approaches out of APPATCH. As shown, Claude-3.5 and GPT-4 achieve the best F1 scores (33.57% and 31.37%) with APPATCH, outperforming the ablated versions. Specifically, Standard Prompting and Zero-shot Completion have the worst performance (1.79%-9.81% and 0.00%-8.39% respectively), showing the effectiveness of APPATCH on the interprocedural samples.

E Data Leakage Assessment

We checked ExtractFix dataset against leakage with following LLM queries:

1. *Please tell the Project, CWE-ID, and CVE description of <CVE-ID>.*

2. *Please tell the CWE-ID and CVE description of <CVE-ID> in <Project>.*

3. *Please tell the patch of <CVE-ID> which has a <CWE-ID> vulnerability in <Project>.*

The results are shown in Table 11. As shown, the LLMs either do not know the answers or provide incorrect answers in most of the cases. This indicates that the data leakage issue is not serious. However, LLMs are largely a black box as we used them in APPATCH, making accurate data leakage assessment very challenging. Thus, we acknowledge that, despite our effort trying to assess the data leakage issues with this dataset, we can not ensure that there was no leakage, nor can we claim that our quantification of the leakage is perfectly accurate. Instead, we took our best effort to assess the leakage, using the queries designed above, as justified as follows. First, we consider that there are five important pieces of the knowledge about how a vulnerability is patched (i.e., the knowledge we are concerned that LLM may already have, when we evaluate APPATCH using the ExtractFix dataset which consists of program samples for which these five pieces of information are provided): (1) *CVE ID*, (2) *project name*, (3) *CVE description*, (4) *CWE ID*, and (5) *the patch*. Then, we design three queries that assess whether an LLM has one or more pieces of such knowledge when given the other pieces. More specifically:

- **Query 1**: we provide the LLM with (1) and ask it about (2), (3), and (4);

- **Query 2**: we provide the LLM with (1) and (2), and ask it about (3) and (4);

- **Query 3**: we provide the LLM with (1), (2), and (4), and ask it about (5).

As seen, with these three queries sent to the LLM in order of Queries 1, 2, 3, we gradually provide the LLM with increasingly more pieces of information, and check if the LLM knows about the rest. This helps us check the LLM against the data leakage with varying levels of difficulty, and put together we believe the LLM's responses can give us a perspective into the data leakage issue. Nevertheless, we could not make bold claims that these queries are the best to assess data leakage about vulnerability patches in general. Moreover, we would like to note the following points: (1) as a comparison, we also test the LLMs with a well-known vulnerability CVE-2021-44228 using the same three queries. The LLMs did successfully answer some of five pieces of information about

Table 8: Composition and statistics of the datasets

Dataset	Project-distribution (project:#CVEs)	CWE-distribution(CWE:#CVEs)	Complexity
Zero-Day	linux:73, git:2, ffmpeg:1, less:1, jasper:2, FreeRDP:3, pytorch:1, Fast-DDS:1, esp-idf:1, openssl:1, suricata:1, RedisBloom:1, tcpdump:1, sngrep:1 jerryscript:2, mynewt-nimble:1, dcmtk:3, libhtp:1	CWE787:17, CWE125:23, CWE190:11, CWE401:11, CWE457:10, CWE476:25	average #patched lines:7.87
ExtractFix	coreutils:2, libjpeg:4, libxml2:2, jasper:1, libtiff:8, binutils-gdb:2, FFmpeg:1	CWE-787:6, CWE-125:7, CWE-190:5, CWE-401:0, CWE-457:1, CWE-476:1	average #patched lines:3.05

Table 9: Code LLMs with APPATCH against the Zero-Day dataset

	SynEq			SemEq			Plausible			Correct		
Model	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**
CodeLlama	0.00%	0.00%	***0.00%***	1.03%	0.42%	***0.60%***	1.03%	0.43%	***0.60%***	2.06%	0.85%	***1.20%***
CodeQwen-1.5	0.00%	0.00%	***0.00%***	4.12%	2.03%	***2.72%***	15.46%	4.93%	***7.47%***	18.55%	6.95%	***10.12%***
DeepSeek-Coder-V2	2.06%	0.68%	***1.02%***	9.28%	3.73%	***5.32%***	6.19%	2.03%	***3.06%***	15.46%	6.44%	***9.09%***

Table 10: Correct reasoning rate on the Zero-Day dataset

Model	Prompt	Correct Rate
GPT-4	Vulnerability Semantics Reasoning	**82.64%**
	Direct Reasoning	73.12%
	No slicing	79.41%
Gemini-1.5	Vulnerability Semantics Reasoning	**63.41%**
	Direct Reasoning	59.62%
	No Slicing	51.97%
Claude-3.5	Vulnerability Semantics Reasoning	**82.64%**
	Direct Reasoning	74.18%
	No Slicing	79.41%
Llama-3.1	Vulnerability Semantics Reasoning	**69.94%**
	Direct Reasoning	50.13%
	No Slicing	64.74%

this CVE correctly, indicating that our queries are helpful to test the data leakage. (2) note that APPATCH's performance evaluation is primarily based on the Zero-Day dataset, and we have compared it with both LLM-based and non-LLM-based baseline patching techniques on this dataset. The reason we use the ExtractFix dataset is mainly for being able to compare with two additional state-of-the-art baselines (ExtractFix [20] and VulnFix [55]).

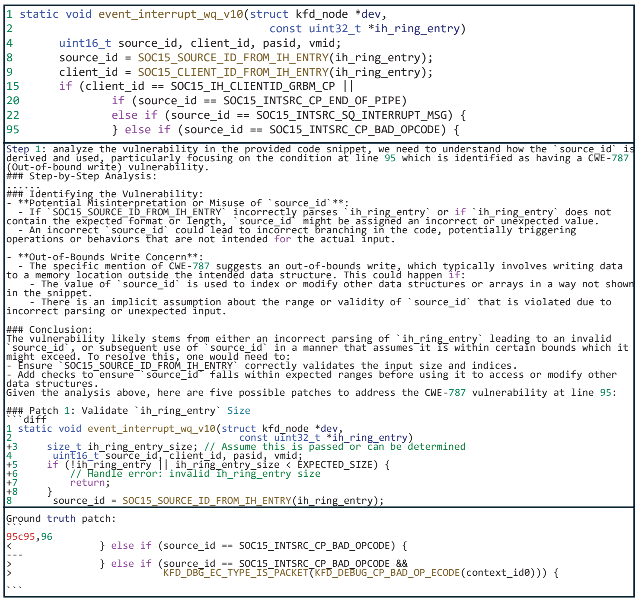
Table 11: Data leakage test results on the ExtractFix dataset with the format "model-does-not-know%, incorrect-answer%, correct-answer%"

Query	GPT-4	Gemini-1.5	Claude-3.5	Llama-3.1
1	100%, 0%, 0%	0%, 100%, 0%	10%, 90%, 0%	25%, 70%, 5%
2	30%, 60%, 10%	5%, 85%, 10%	55%, 45%, 0%	5%, 90%, 5%
3	100%, 0%, 0%	25%, 75%, 0%	100%, 0%, 0%	70%, 30%, 0%

F More End-to-end Experiment Results

Table 13 shows the detailed end-to-end integration results with CodeQL against the Zero-Day dataset. As shown, APPATCH achieves 23.85% F1 with fully automated method and 27.29% with realistic method. In contrast, the baselines approaches and models have much less effectiveness, indicating that APPATCH can be realistically used in practice.

G More Details on Case Study Examples



~~~~
1 static void event_interrupt_wq_v10(struct kfd_node *dev,
2                                         const uint32_t *ih_ring_entry)
4       uint16_t source_id, client_id, pasid, vmid;
8       source_id = SOC15_SOURCE_ID_FROM_IH_ENTRY(ih_ring_entry);
9       client_id = SOC15_CLIENT_ID_FROM_IH_ENTRY(ih_ring_entry);
15      if (client_id == SOC15_IH_CLIENTID_GRBM_CP ||
20              if (source_id == SOC15_INTSRC_CP_END_OF_PIPE)
22              else if (source_id == SOC15_INTSRC_SQ_INTERRUPT_MSG) {
95              } else if (source_id == SOC15_INTSRC_CP_BAD_OPCODE) {
~~~~

~~~~
Step 1: analyze the vulnerability in the provided code snippet, we need to understand how the `source_id` is
derived and used, particularly focusing on the condition at line 95 which is identified as having a CWE-787
(Out-of-bound write) vulnerability.
### Step-by-Step Analysis:
......
### Identifying the Vulnerability:
- **Potential Misinterpretation or Misuse of `source_id`**:
  - If `SOC15_SOURCE_ID_FROM_IH_ENTRY` incorrectly parses `ih_ring_entry` or if `ih_ring_entry` does not
contain the expected format or length, `source_id` might be assigned an incorrect or unexpected value.
  - An incorrect `source_id` could lead to incorrect branching in the code, potentially triggering
operations or behaviors that are not intended for the actual input.

- **Out-of-Bounds Write Concern**:
  - The specific mention of CWE-787 suggests an out-of-bounds write, which typically involves writing data
to a memory location outside the intended data structure. This could happen if:
    - The value of `source_id` is used to index or modify other data structures or arrays in a way not shown
in the snippet.
    - There is an implicit assumption about the range or validity of `source_id` that is violated due to
incorrect parsing or unexpected input.

### Conclusion:
The vulnerability likely stems from either an incorrect parsing of `ih_ring_entry` leading to an invalid
`source_id`, or subsequent use of `source_id` in a manner that assumes it is within certain bounds which it
might exceed. To resolve this, one would need to:
- Ensure `SOC15_SOURCE_ID_FROM_IH_ENTRY` correctly validates the input size and indices.
- Add checks to ensure `source_id` falls within expected ranges before using it to access or modify other
data structures.
Given the analysis above, here are five possible patches to address the CWE-787 vulnerability at line 95:

### Patch 1: Validate `ih_ring_entry` Size
```diff
1 static void event_interrupt_wq_v10(struct kfd_node *dev,
2                                        const uint32_t *ih_ring_entry)
+3     size_t ih_ring_entry_size; // Assume this is passed or can be determined
4       uint16_t source_id, client_id, pasid, vmid;
+5     if (!ih_ring_entry || ih_ring_entry_size < EXPECTED_SIZE) {
+6         // Handle error: invalid ih_ring_entry size
+7         return;
+8     }
8       source_id = SOC15_SOURCE_ID_FROM_IH_ENTRY(ih_ring_entry);
~~~~

~~~~
Ground truth patch:
```
95c95,96
<               } else if (source_id == SOC15_INTSRC_CP_BAD_OPCODE) {
---
>               } else if (source_id == SOC15_INTSRC_CP_BAD_OPCODE &&
>                          KFD_DBG_EC_TYPE_IS_PACKET(KFD_DEBUG_CP_BAD_OP_ECODE(context_id0))) {
```
~~~~

Figure 10: A failure case where the vulnerability is incorrectly identified.

We conduct case studies on the APPATCH's failure cases to investigate the symptoms and root causes. As discussed in Section 6.3, we show two examples for the failure symptoms and root causes.

Figure 10 shows an example where the failed patching attempt indicates that the original vulnerability was not accurately identified or addressed. The symptom was incorrectly diagnosed as an out-of-bounds write (CWE-787) related to the 'source_id' variable, leading to proposed patches that focused on validating and constraining this value. However, the root cause appears to be a logic error or insufficient condition checking in the original code. The actual fix involves adding an additional condition to check if the context ID corresponds to a specific packet type, suggesting that the vulnerability was

Table 12: Detailed results of APPATCH against the interprocedural samples from the Zero-Day dataset

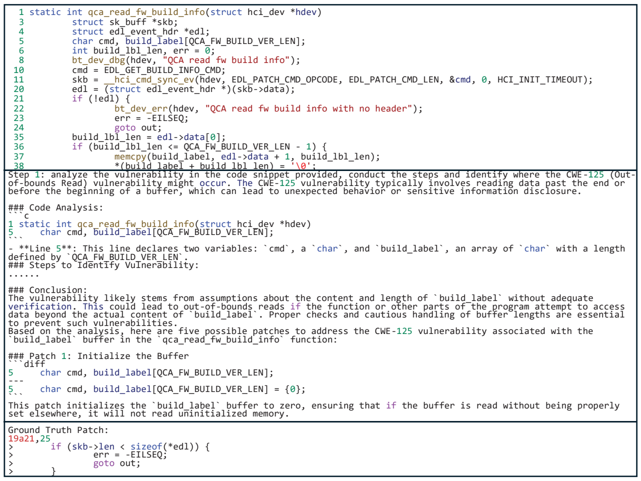
Model	Approach	Zero-Day Dataset											
		SynEq			**SemEq**			**Plausible**			**Correct**		
		Recall	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**
GPT-4	APPATCH	4.76%	1.11%	***1.80%***	9.52%	4.44%	***6.06%***	38.10%	21.11%	***27.17%***	38.10%	26.67%	***31.37%***
	No Validation	4.76%	0.90%	*1.52%*	14.29%	5.41%	*7.84%*	38.10%	18.92%	*25.28%*	38.10%	25.23%	*30.35%*
	No Slicing	0.00%	0.00%	*0.00%*	9.52%	1.82%	*3.05%*	33.33%	16.36%	*21.95%*	33.33%	18.18%	*23.53%*
	Random Exemplars	4.76%	0.90%	*1.52%*	9.52%	1.80%	*3.03%*	28.57%	12.61%	*17.50%*	33.33%	15.32%	*20.99%*
	Manual Exemplars	0.00%	0.00%	*0.00%*	9.52%	2.73%	*4.24%*	33.33%	14.55%	*20.25%*	38.10%	17.27%	*23.77%*
	Direct Reasoning	0.00%	0.00%	*0.00%*	4.76%	0.90%	*1.52%*	38.10%	12.61%	*18.95%*	38.10%	13.51%	*19.95%*
	Standard Prompting	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	19.05%	6.60%	9.81%	19.05%	6.60%	9.81%
	Zero-shot Completion	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.52%	4.72%	6.31%	9.52%	4.72%	6.31%
Gemini-1.5	APPATCH	0.00%	0.00%	***0.00%***	9.52%	2.02%	***3.33%***	28.57%	12.12%	***17.02%***	28.57%	14.14%	***18.91%***
	No Validation	0.00%	0.00%	*0.00%*	9.52%	2.02%	*3.33%*	28.57%	11.32%	*16.22%*	28.57%	13.34%	*18.18%*
	No Slicing	0.00%	0.00%	*0.00%*	9.52%	2.06%	*3.39%*	14.28%	10.31%	*11.98%*	23.81%	12.37%	*16.28%*
	Random Exemplars	0.00%	0.00%	*0.00%*	9.52%	2.08%	*3.42%*	19.05%	9.38%	*12.57%*	23.81%	11.46%	*15.47%*
	Manual Exemplars	0.00%	0.00%	*0.00%*	9.52%	2.02%	*3.33%*	23.81%	9.09%	*13.16%*	23.81%	11.11%	*15.15%*
	Direct Reasoning	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	23.81%	7.61%	11.54%	23.81%	7.62%	11.54%
	Standard Prompting	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	4.76%	1.10%	1.79%	4.76%	1.10%	1.79%
	Zero-shot Completion	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Claude-3.5	APPATCH	4.76%	0.93%	***1.55%***	19.05%	3.70%	***6.20%***	47.62%	21.30%	***29.43%***	47.62%	25.93%	***33.57%***
	No Validation	4.76%	0.90%	*1.52%*	19.05%	3.60%	*6.06%*	47.62%	20.72%	*28.88%*	47.62%	25.23%	*32.98%*
	No Slicing	0.00%	0.00%	*0.00%*	9.52%	1.98%	*3.28%*	38.10%	19.80%	*26.06%*	38.10%	21.78%	*27.72%*
	Random Exemplars	0.00%	0.00%	*0.00%*	4.76%	0.90%	*1.52%*	42.86%	18.92%	*26.25%*	47.62%	19.82%	*27.99%*
	Manual Exemplars	0.00%	0.00%	*0.00%*	14.29%	2.70%	*4.55%*	42.86%	16.22%	*23.53%*	42.86%	18.92%	*26.25%*
	Direct Reasoning	0.00%	0.00%	0.00%	9.52%	1.89%	3.15%	38.10%	17.92%	24.38%	42.86%	19.81%	27.10%
	Standard Prompting	0.00%	0.00%	0.00%	4.76%	1.96%	2.78%	9.52%	4.90%	6.47%	14.29%	6.86%	9.27%
	Zero-shot Completion	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	14.29%	5.94%	8.39%	14.29%	5.94%	8.39%
Llama-3.1	APPATCH	0.00%	0.00%	***0.00%***	9.52%	2.35%	***3.77%***	38.10%	15.29%	***21.82%***	38.10%	17.65%	***24.12%***
	No Validation	0.00%	0.00%	*0.00%*	9.52%	1.80%	*3.03%*	38.10%	12.61%	*18.95%*	38.10%	14.41%	*20.91%*
	No Slicing	0.00%	0.00%	*0.00%*	4.76%	3.85%	*4.26%*	9.52%	19.23%	*12.74%*	14.29%	23.08%	*17.65%*
	Random Exemplars	0.00%	0.00%	*0.00%*	4.76%	1.98%	*2.80%*	19.05%	8.91%	*12.14%*	23.81%	10.89%	*14.95%*
	Manual Exemplars	0.00%	0.00%	*0.00%*	4.76%	2.56%	*3.33%*	19.05%	12.82%	*15.33%*	19.05%	15.38%	*17.02%*
	Direct Reasoning	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*28.57%*	*10.81%*	*15.68%*	*28.57%*	*10.81%*	*15.68%*
	Standard Prompting	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*9.52%*	*3.13%*	*4.71%*	*9.52%*	*3.13%*	*4.71%*
	Zero-shot Completion	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*9.52%*	*2.70%*	*4.21%*	*9.52%*	*2.70%*	*4.21%*

Table 13: Detailed end-to-end results with vulnerability analyzer CodeQL against the Zero-Day dataset

Method	Model	Approach	Zero-Day Dataset											
			SynEq			**SemEq**			**Plausible**			**Correct**		
			Recall	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**
Fully Automated	Claude-3.5	APPATCH	1.03%	0.33%	***0.50%***	12.37%	7.21%	***9.11%***	26.80%	10.82%	***15.42%***	34.02%	18.36%	***23.85%***
		Manual Exemplars	1.03%	0.32%	*0.49%*	7.22%	5.18%	*6.03%*	15.46%	5.83%	*8.46%*	22.68%	11.33%	*15.11%*
		Standard Prompting	0.00%	0.00%	0.00%	8.25%	5.25%	6.41%	11.34%	5.90%	7.76%	16.49%	11.15%	13.30%
		Zero-shot Completion	0.00%	0.00%	0.00%	2.06%	0.78%	1.13%	14.43%	8.27%	10.51%	15.46%	9.04%	11.41%
	VulRepair	-	2.06%	0.47%	0.77%	13.40%	8.47%	10.38%	1.03%	0.24%	0.38%	16.49%	9.18%	11.79%
	Getafix	-	5.15%	5.26%	5.21%	0.00%	0.00%	0.00%	1.03%	0.44%	0.62%	6.19%	5.70%	5.93%
Realistic	Claude-3.5	APPATCH	1.03%	0.25%	***0.40%***	19.59%	7.71%	***11.07%***	29.90%	12.44%	***17.57%***	41.24%	20.40%	***27.29%***
		Manual Exemplars	0.00%	0.00%	*0.00%*	15.46%	6.05%	*8.70%*	15.46%	5.33%	*7.92%*	25.77%	11.38%	*15.79%*
		Standard Prompting	1.03%	0.28%	0.44%	19.59%	9.47%	12.77%	13.40%	6.41%	8.67%	24.74%	16.16%	19.55%
		Zero-shot Completion	0.00%	0.00%	0.00%	2.06%	0.78%	1.13%	14.43%	8.27%	10.51%	15.46%	9.04%	11.41%
	VulRepair	-	2.06%	0.47%	0.77%	13.40%	8.47%	10.38%	1.03%	0.24%	0.38%	16.49%	9.18%	11.79%
	Getafix	-	5.15%	5.26%	5.21%	0.00%	0.00%	0.00%	1.03%	0.44%	0.62%	6.19%	5.70%	5.93%

more likely related to improper input validation or incorrect handling of certain packet types, rather than a straightforward buffer overflow issue.

Figure 11 shows an example where failed patching attempts primarily focused on addressing potential issues with the ‘build_label‘ array’s initialization and usage, but they missed the root cause of the vulnerability. The actual problem stems from insufficient bounds checking when accessing data from the ‘skb‘ (socket buffer) structure. The ground truth patch reveals that the vulnerability is caused by inadequate validation of the ‘skb->len‘ against the expected data sizes at various points in the function. This allows for potential out-of-bounds reads when accessing ‘edl‘ structure members and data. The correct fix involves adding multiple checks to ensure that the ‘skb‘ contains enough data before accessing its contents, and dynamically allocating memory for ‘build_label‘ instead of using a fixed-size array, which addresses both buffer overflow and potential memory leaks.



```
 1 static int qca_read_fw_build_info(struct hci_dev *hdev)
 3         struct sk_buff *skb;
 4         struct edl_event_hdr *edl;
 5         char cmd, build_label[QCA_FW_BUILD_VER_LEN];
 6         int build_lbl_len, err = 0;
 8         bt_dev_dbg(hdev, "QCA read fw build info");
10         cmd = EDL_GET_BUILD_INFO_CMD;
11         skb = __hci_cmd_sync_ev(hdev, EDL_PATCH_CMD_OPCODE, EDL_PATCH_CMD_LEN, &cmd, 0, HCI_INIT_TIMEOUT);
20         edl = (struct edl_event_hdr *)(skb->data);
21         if (!edl) {
22                 bt_dev_err(hdev, "QCA read fw build info with no header");
23                 err = -EILSEQ;
24                 goto out;
35         build_lbl_len = edl->data[0];
36         if (build_lbl_len <= QCA_FW_BUILD_VER_LEN - 1) {
37                 memcpy(build_label, edl->data + 1, build_lbl_len);
38                 *(build_label + build_lbl_len) = '\0';
```

~~~
Step 1: analyze the vulnerability in the code snippet provided, conduct the steps and identify where the CWE-125 (Out-
of-bounds Read) vulnerability might occur. The CWE-125 vulnerability typically involves reading data past the end or
before the beginning of a buffer, which can lead to unexpected behavior or sensitive information disclosure.

### Code Analysis:
```c
1 static int qca_read_fw_build_info(struct hci_dev *hdev)
5     char cmd, build_label[QCA_FW_BUILD_VER_LEN];
~~~

- **Line 5**: This line declares two variables: `cmd`, a `char`, and `build_label`, an array of `char` with a length
defined by `QCA_FW_BUILD_VER_LEN`.
### Steps to Identify Vulnerability:
......

### Conclusion:
The vulnerability likely stems from assumptions about the content and length of `build_label` without adequate
verification. This could lead to out-of-bounds reads if the function or other parts of the program attempt to access
data beyond the actual content of `build_label`. Proper checks and cautious handling of buffer lengths are essential
to prevent such vulnerabilities.
Based on the analysis, here are five possible patches to address the CWE-125 vulnerability associated with the
`build_label` buffer in the `qca_read_fw_build_info` function:

### Patch 1: Initialize the Buffer
```diff
5     char cmd, build_label[QCA_FW_BUILD_VER_LEN];
---
5     char cmd, build_label[QCA_FW_BUILD_VER_LEN] = {0};
```

This patch initializes the `build_label` buffer to zero, ensuring that if the buffer is read without being properly
set elsewhere, it will not read uninitialized memory.
```

```
Ground Truth Patch:
19a21,25
>         if (skb->len < sizeof(*edl)) {
>                 err = -EILSEQ;
>                 goto out;
>         }
```

Figure 11: A failure case where the patch has insufficient or incorrect code modification.
```