Automated Software Vulnerability Patching using Large Language Models

Motivation

- Timely and effective vulnerability patching is essential for cybersecurity defense
- Current approaches struggle to generate valid and correct patches for real-world vulnerabilities
- The authors propose to use LLM to automate vul. patching
    - No test input/exploit evidence
    - No model training

Key Challenges

1) For automated patching, prompting would also need to be automated
2) We need exemplars for effective prompting, but how to automatically get the best exemplars
3) Real-world vulnerabilities have cross-file dependencies, how to accommodate the large code context
4) LLM can hallucinate, bad patching can be worse than not patching

Motivating Example

Vulnerable code for CVE-2024-24188

Vulnerable statement is at line 24

```
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 }
```

Data Dependency

Control Dependency

GPT4 + Standard Prompt

The vulnerability is correctly identified, but it does not analyze the actual buffer size of p

Q: Given the following code: <code in Figure 1> which has a CWE-787 vulnerability at line 24, please generate the patch for the vulnerability.

A:The CWE-787 vulnerability in the provided code slice 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. 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);
```

GPT4+ SOTA Approach

In the prev SOTA approach, the vulnerable statement is removed and LLM is instructed to complete the code

Still the vulnerability is not patched

```
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);
        ......
```
````

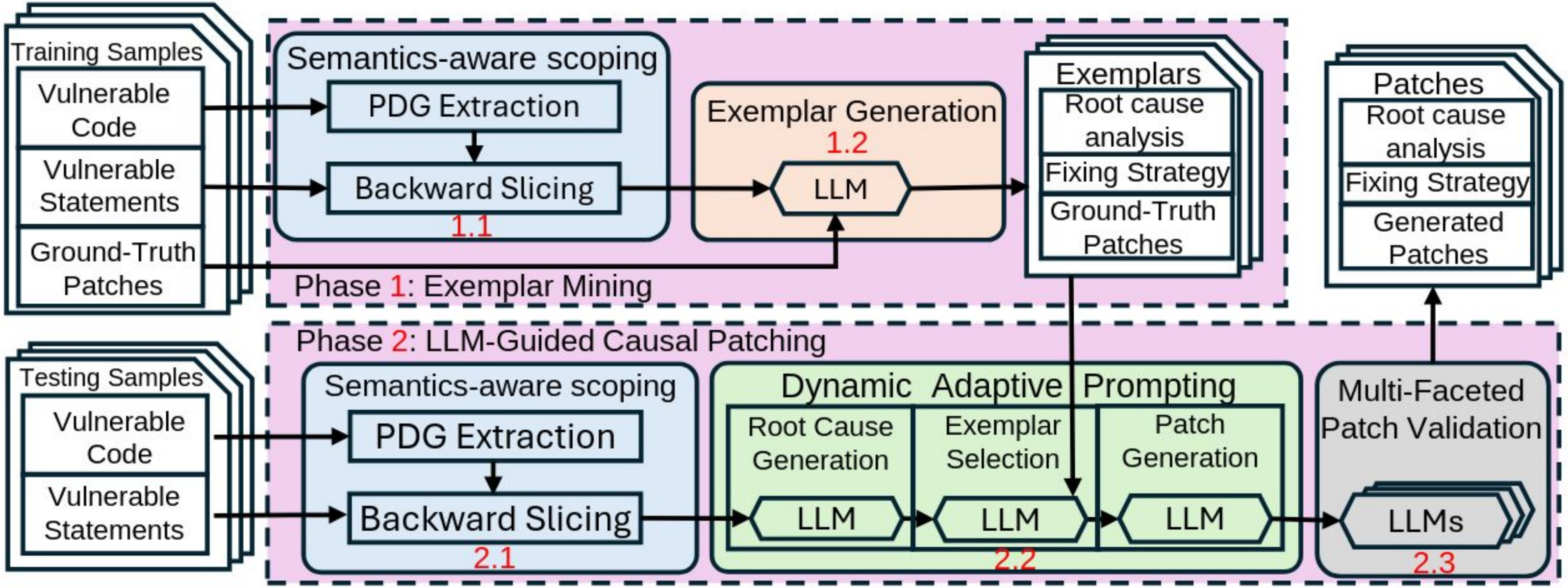
LLMPATCH: Open source tool

- Implement LLMPATCH based on four latest and most powerful LLMs: GPT4, Gemini, Claude3, and Llama3.
- Key results on vulnerability patching
  - LLMPATCH: 44.91%-57.18% F1
  - Baseline Prompting: 3.54%-28.74% F1
  - Prev SOTA: 20.58%-34.58% F1
- LLMPATCH is efficient: 37.148-50.209s
- LLMPATCH is inexpensive: 5,684-6,802 tokens per patch (less than 1 $)
- On the latest 11 vulnerabilities, LLMPATCH successfully patches 7 while the baselines only patches 0-3 vulnerabilities.

Key aspects of LLMPATCH

1) Narrows down the scope of analysis to only the relevant subset of the program via a step called semantics-aware scoping. (Challenge 3)
2) Elicits the LLM to identify the vulnerability's root cause and uses it to mine exemplars in a step called dynamic adaptive prompting. (Challenge 2)
3) With the adaptively chosen exemplars, LLMPATCH forms the patching prompt automatically. (Challenge 1)
4) LLMPATCH consults an ensemble of LLMs to cross-validate the candidate patches. (Challenge 4)

LLMPATCH OVERVIEW



Semantic Aware Scoping

- LLMPATCH extracts the essential code for vulnerability patching from the vulnerable code sample.
    - Small portion of code are critical statements, named as vulnerability semantics. Enable LLM to be more effective
    - LLM struggle with long text
- Vulnerability semantics: Aspects of a program's code that contribute to its vulnerabilities
- Need to understand control and data flow dependencies
- Parse the source code into program dependence graphs
- Iteratively traverse this graph to collect all related parts of code

Motivating Example

Vulnerable code for CVE-2024-24188

Vulnerable statement is at line 24

```
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 }
```

Data Dependency

Control Dependency

Root Cause Analysis

Given a vulnerability semantics slice, the CWE ID, and the vulnerable statement, LLMPATCH first prompts the LLMs to generate the root cause

Q: Given the following code slice

```
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;
6     cp = (char*)str;
7     while (*cp) {
8         if (ch)
9             cp = Jsi_Strchr(cp,ch);
11            while (*cp && !isspace(*cp))
12                cp++;
13        if (cp == NULL || *cp == 0)
14            break;
15        cp++;
16        cnt++;
19    argv = (char**)Jsi_DSValue(dStr);
22    p = (char*)&(argv[cnt+2]);
24    Jsi_Strcpy(p, str);
```

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 say "Please provide the summary of the function <func_name>".

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.

Fixing Strategy

With the ground-truth patches provided, LLMs are capable of generating the correct reasoning steps

Conduct a preliminary experiment on the real-world samples from our collected BigVul+CVEFixes dataset

Manual inspection on the generated exemplars by GPT-4, 92.98% of the reasoning steps are correct

Q: Given the following code slice <vulnerability semantics slice>, which has a <CWE-ID> vulnerability at line <vulnerable statement line>, the patch is <ground-truth patch>, please provide the reasoning steps to generate this patch.

Exemplar Selection

The time complexity is O(mn) where m is the number of training samples and n is the number of testing samples

Upto 8 exemplars are selected based on previous work in prompting

Q: Does the following two vulnerabilities share similar root causes:

<training sample root cause>

<testing sample root cause>

Please simply answer yes or no.

Patch Generation

Multiple patches are generated for a given input

To avoid re-computing root cause tokens, it is already added to the prompt from previous step

<selected exemplars>

Q: Given the following code slice: <slice code> which has a <CWE-ID> vulnerability at line: <vulnerable line>, please generate five possible patches for the vulnerability.

A: The patch can be done in three steps. Step 1. <root cause analysis>

Benchmark

- Start with PatchDB (12K) and CVEFixes (4K)
- Select the samples with the most popular CWEs in C languages.
- Collect 306 vulnerability fixing samples including 93 CWE-787, 45 CWE-125, 58 CWE-190, 21 CWE-401, 19 CWE-457, and 70 CWE-476.
- Randomly split 200 samples for training and the remaining 106 samples are used for testing

Evaluation Metric

- recall = #fixed samples / #testing sample
- precision = #correct patches / #generated patches

Main Results

Technique/Approach	SynEq			SemEq			Plausible			Correct		
	Recall	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**	**Recall**	**Prec**	**F1**
LLMPatch (GPT-4)	45.28%	20.14%	***27.88%***	43.39%	16.31%	***23.71%***	33.02%	13.43%	***19.09%***	66.98%	49.88%	***57.18%***
Standard Prompting (GPT-4)	21.69%	6.96%	*10.55%*	17.92%	5.62%	*8.55%*	21.69%	9.66%	*13.37%*	40.56%	22.24%	*28.74%*
Zero-shot Completion (GPT-4)	12.26%	3.72%	*5.71%*	5.66%	1.18%	*1.95%*	18.87%	12.94%	*15.35%*	24.52%	17.84%	*20.66%*
LLMPatch (Gemini)	42.45%	20.33%	***27.49%***	37.73%	18.13%	***24.49%***	29.24%	12.36%	***17.37%***	57.54%	50.82%	***53.97%***
Standard Prompting (Gemini)	9.43%	2.85%	*4.38%*	5.66%	2.59%	*3.56%*	7.54%	3.11%	*4.41%*	18.86%	8.57%	*11.78%*
Zero-shot Completion (Gemini)	1.88%	1.33%	*1.56%*	0.00%	0.00%	*0.00%*	1.88%	2.00%	*1.94%*	3.77%	3.33%	*3.54%*
LLMPatch (Claude3)	30.18%	11.91%	***17.08%***	33.02%	16.83%	***22.30%***	20.75%	9.32%	***12.86%***	54.72%	38.08%	***44.91%***
Standard Prompting (Claude3)	14.15%	4.75%	*7.11%*	9.43%	3.00%	*4.55%*	12.26%	7.75%	*9.49%*	24.52%	15.50%	*18.99%*
Zero-shot Completion (Claude3)	3.77%	0.79%	*1.31%*	3.77%	0.79%	*1.31%*	16.98%	10.09%	*12.66%*	18.87%	11.68%	*14.43%*
LLMPatch (Llama3)	29.24%	27.61%	***28.41%***	30.18%	23.33%	***26.32%***	21.69%	17.61%	***19.44%***	39.62%	68.57%	***50.22%***
Standard Prompting (Llama3)	15.09%	5.14%	*7.67%*	14.15%	4.86%	*7.23%*	18.86%	9.71%	*12.82%*	33.96%	19.71%	*24.94%*
Zero-shot Completion (Llama3)	5.66%	1.43%	*2.28%*	3.77%	1.43%	*2.07%*	9.43%	4.08%	*5.69%*	13.20%	6.93%	*9.09%*
VulRepair	50.00%	16.03%	*24.28%*	13.21%	2.83%	*4.66%*	17.92%	5.84%	*8.82%*	57.54%	24.71%	*34.58%*
Getafix	24.52%	7.15%	*11.08%*	18.86%	6.94%	*10.15%*	3.77%	0.84%	*1.38%*	33.02%	14.94%	*20.58%*

Contribution Of Components

		SynEq			SemEq			Plausible			Correct		
Model	Approach	Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1	Recall	Prec	F1
GPT-4	LLMPatch	45.28%	20.14%	***27.88%***	43.39%	16.31%	***23.71%***	33.02%	13.43%	***19.09%***	66.98%	49.88%	***57.18%***
	No Validation	45.28%	18.87%	*26.64%*	43.39%	15.28%	*22.60%*	33.02%	12.58%	*18.22%*	66.98%	46.74%	*55.06%*
	No Slicing	43.39%	15.28%	*22.60%*	29.24%	10.56%	*15.52%*	25.47%	9.88%	*14.25%*	58.49%	35.73%	*44.36%*
	Random Exemplars	40.56%	13.76%	*20.55%*	32.07%	11.61%	*17.05%*	25.47%	8.60%	*12.86%*	62.26%	33.97%	*43.96%*
	Fixed Exemplars	33.96%	11.76%	*17.47%*	23.58%	8.82%	*12.84%*	26.41%	9.21%	*13.66%*	55.66%	29.80%	*38.82%*
Gemini	LLMPatch	42.45%	20.33%	***27.49%***	37.73%	18.13%	***24.49%***	29.24%	12.36%	***17.37%***	57.54%	50.82%	***53.97%***
	No Validation	42.45%	15.25%	*22.45%*	37.73%	13.61%	*20.00%*	29.24%	9.27%	*14.08%*	57.54%	38.14%	*45.88%*
	No Slicing	12.26%	13.33%	*12.78%*	8.49%	11.33%	*9.71%*	8.49%	10.67%	*9.45%*	17.92%	35.33%	*23.78%*
	Random Exemplars	29.24%	8.91%	*13.65%*	28.30%	7.92%	*12.38%*	16.04%	4.95%	*7.56%*	50.00%	21.78%	*30.34%*
	Fixed Exemplars	23.58%	6.53%	*10.23%*	16.98%	4.35%	*6.93%*	19.81%	5.74%	*8.90%*	40.56%	16.63%	*23.59%*
Claude3	LLMPatch	30.18%	11.91%	***17.08%***	33.02%	16.83%	***22.30%***	20.75%	9.32%	***12.86%***	54.72%	38.08%	***44.91%***
	No Validation	30.18%	10.95%	*16.07%*	33.02%	15.47%	*21.07%*	20.75%	8.57%	*12.13%*	54.72%	35.00%	*42.69%*
	No Slicing	28.30%	10.63%	*15.45%*	24.52%	10.88%	*15.08%*	29.24%	13.92%	*18.86%*	40.56%	35.44%	*37.83%*
	Random Exemplars	29.24%	9.14%	*13.93%*	30.18%	14.04%	*19.16%*	27.35%	10.21%	*14.87%*	47.16%	33.40%	*39.11%*
	Fixed Exemplars	28.30%	10.00%	*14.78%*	29.24%	9.21%	*14.01%*	31.13%	11.17%	*16.44%*	53.77%	30.39%	*38.83%*
Llama3	LLMPatch	29.24%	27.61%	***28.41%***	30.18%	23.33%	***26.32%***	21.69%	17.61%	***19.44%***	39.62%	68.57%	***50.22%***
	No Validation	29.24%	22.31%	*25.31%*	30.18%	18.84%	*23.21%*	21.69%	14.23%	*17.18%*	39.62%	55.38%	*46.19%*
	No Slicing	16.03%	8.00%	*10.67%*	20.75%	15.00%	*17.41%*	9.43%	7.33%	*8.25%*	27.35%	30.33%	*28.76%*
	Random Exemplars	35.84%	16.92%	*22.99%*	29.24%	12.05%	*17.06%*	20.75%	9.74%	*13.26%*	48.11%	38.71%	*42.91%*
	Fixed Exemplars	2.83%	13.84%	*4.69%*	5.67%	12.31%	*7.75%*	3.77%	6.15%	*4.67%*	7.54%	32.30%	*12.23%*

Zero-day Vulnerability Patching

CWE ID	Project	CVE ID	LLMPatch	Standard Prompting	Zero-shot Completion	VulRepair	Getafix
CWE-787	jsish	2024-24188	✓	✗	✗	✗	✗
CWE-787	libcoap	2024-0962	✗	✗	✗	✗	✗
CWE-787	jasper	2023-51257	✓	✗	✗	✓	✗
CWE-401	libming	2024-24150	✓	✓	✓	✗	✗
CWE-401	libming	2024-24149	✓	✓	✓	✗	✗
CWE-125	libcoap	2023-30362	✓	✗	✗	✗	✗
CWE-125	Espruino	2024-25201	✗	✗	✗	✗	✗
CWE-125	gpac	2024-0322	✗	✗	✗	✗	✗
CWE-476	cJSON	2023-50472	✓	✗	✗	✓	✗
CWE-476	RedisGraph	2023-47003	✗	✗	✗	✗	✗
CWE-190	plutosvg	2023-44709	✓	✓	✗	✗	✗
Correct Rate			7/11	3/11	2/11	2/11	0/11

Scientific Peer Reviewer: Pankayaraj

Paper Summary

- This paper presents the idea of automating the patch generation for a given vulnerability in the code.
- Here they design the patching via an automated adaptive prompting technique where they first use a labeled dataset with vulnerabilities and patches to create examples in an automated manner. Here the vulnerable lines are complemented with the dependency lines which gives a full context of the vulnerability and then the ground truth patch is suggested for it.
- Later when given a vulnerable code and the vulnerability line they form a set of dependency lines for the vulnerability in a similar manner as before and then get a summary of the root cause.
- Then they use the summary as a way to match suitable examples and use in context learning to provide them as examples and ask the LLM to patch the vulnerabilities.
- Here they use an ensemble of LLMs to validate the vulnerability patch to avoid hallucination

Technical Correctness:

2. Minor Issues

For a report this paper is technically sound.

Technical Comments

- **Use of same LLM for reasoning generation and patching.**
  - For both the exemplar creation (reasoning step for a given patch) and the patching step they use a single LLMs.
  - Given that LLMs are shown to favor their own responses (LLM Evaluators Recognize and Favor Their Own Generations, 2024) and both the reasoning and the patching are done by the same LLMs I would like to see if this type of behavior will affect negatively affect the patching.
  - For instance if the given reasoning wrong but it came from the same LLM (GPT4) will it ignore this as opposed to if the reasoning came from another LLM (eg Gemini) or else regardless of which LLM has generated the reasoning given the context the patching LLM will follow through with that reasoning. Given that in the paper they have stated that by manual checking found 92% of the reasoning is correct I would like to see this above mentioned setting for the examples for which the reasoning was wrong.

Technical Comments

- **Complexity mitigation with RAGs**
  - Though out of the scope of this paper the exemplar selection method is done via O(nm) LLM queries which is expensive. But this process does resemble the process of a RAG system. Can a traditional RAG retriever be used to select relevant root cause samples. This can reduce the complexity especially in practical systems the examples will be huge. If not fine tuning a pre-trained retriever for the root cause retrieval is an interesting future direction
- **Validation of training set reasoning**
  - Why not use multi facet validation to improve the reasoning steps in the train set. Given train set is only created once it would make more sense to use validation to get a better accuracy.

Decision: Accept
(with minor changes)

Archaeologist

Yang Jeffrey Fan Chiang

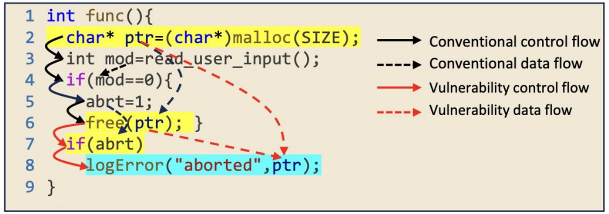
Previous work motivates current paper

Yu Nong
Washington State University
yu.nong@wsu.edu

Chain-of-Thought Prompting of Large Language Models for Discovering and Fixing Software Vulnerabilities

2024 Feb.

- Same author, seems like a previous version of this work
- 3 tasks:
  - 1) vulnerability identification: binary classification (CWE-xxx or CWE-yyy)
  - 2) vulnerability discovery: multiclass classification (which CWE(s) does it have)
  - 3) vulnerability patching: given vulnerable code: <vulnerable code> <vulnerable line> -> generate patch
- Method: Vulnerability-Semantics-guided Prompting (VSP)
  - CoT
  - Highlight vulnerability semantics (not pointing out how this is done)
  - Few Shots (manually provide few shots)



(a) A use-after-free and null-pointer-dereference vulnerability sample with vulnerability semantics marked, where the (possible) vulnerable statement is marked as cyan, the contexts are marked as yellow, and the control-flow and data-flow relationships are marked with arrows.

Current paper

What's the difference

- **Semantics-Aware Scoping: algorithm for automatically extracting vulnerability semantics**
- **Mining exemplars from known patches to build an exemplar database**
- **Dynamically selecting the most appropriate exemplars (semantically similar)**
- **More LLMs**

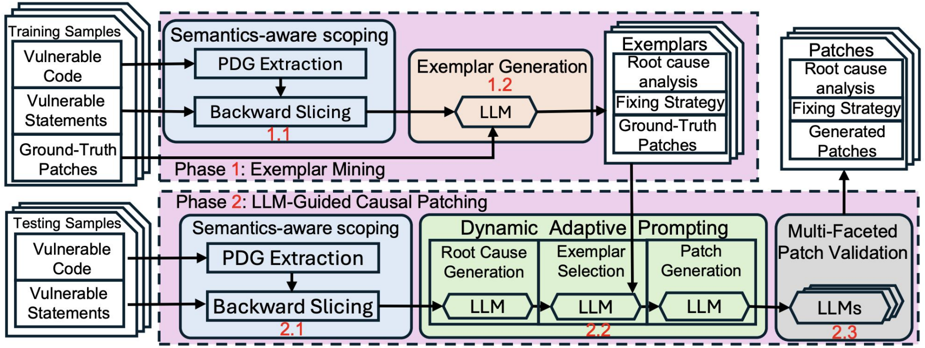


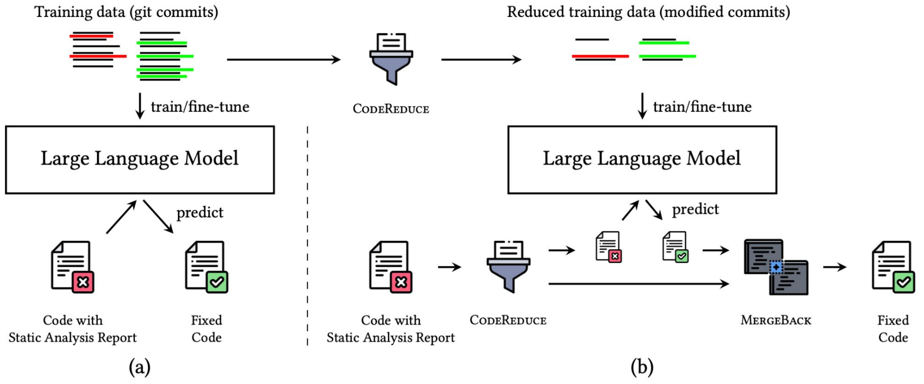
Figure 4. Overview of LLMPatch

Similar work

- No papers have cited the previous one
- DeepCode AI Fix: Fixing Security Vulnerabilities with Large Language Models
  - Propose security and semantic code fixes dataset (with both vulnerabilities and bugs)
  - CodeReduced (extract and simplify code)
    - Hierarchical Delta Debugging (HDD)
    - Reducing the size of a program while preserving a specific property, (e.g.compiler bug, static analysis alarm)
  - 2 scenarios: fine-tune/few-shot
  - Few-shot
    - Random Selection of exemplar

Martin Vechev
martin.vechev@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

2024 Feb.



Archaeologist

Zeying Zhu

Previous work motivates current paper

- **Zero-shot code completion**: "Examining Zero-Shot Vulnerability Repair with Large Language Models"
  - Remove the vulnerable code and lets LLMs complete the vulnerable parts
- **Standard prompting**: "LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?)"
  - Directly ask LLM to patch the vulnerability with code line number and CWE ID

- Both have no code analysis process and thus cannot solve out-of-bound vulnerability.
- LLMs provide incorrect and unfaithful reasoning in automated vulnerability repair without step-by-step guidance.

Solutions from LLMPatch

- Semantics-aware scoping for providing LLMs with code slices only relevant to vulnerabilities
- Chain-of-Thought prompting with exemplars from known patches
  - Finding root cause
  - Fixing strategy
  - Patch generation
- Build an exemplar database automatically for adaptive prompting

Subsequent/Concurrent Work

- Not cited yet but similar work concurrently:
- *Code Vulnerability Repair with Large Language Model using Context-Aware Prompt Tuning*

- Proposing similar findings of **context-aware prompting** with **security contexts** and **code contexts** in the prompt
- Specific on buffer overflow vulnerabilities using GitHib Copilot.
  - Security context: disclosing vulnerability existence to LLM and disclosing CWE details
  - Code context: buffer identification, bound selection, etc.

Industry Practitioner

Aditya Ranjan

AI Assisted CI/CD Pipelines

- A major fintech company developing secure online banking and payment systems.
- Extremely sensitive financial data and transactions for millions of customers
- Any security vulnerability could have severe consequences
  - Financial losses
  - Regulatory penalties
  - Reputation damage
- Security team is struggling with the large volume of potential vulnerabilities that need to be addressed

Advantages

- Increased speed and scale in vulnerability analysis and patching
- 24/7 operation capability
- Improved consistency in patch generation
- Cost efficiency in the long run
  - Utilization of off-the-shelf LLMs without costly fine-tuning
- Seamless integration with existing CI/CD pipelines
- Enhanced developer productivity by automating common vulnerability patching

Disadvantages

- Potential over-reliance on automated systems for critical security functions
- Risk of generated patches introducing new bugs or incompletely addressing vulnerabilities
- Current performance limitations (44.91%-57.18% F1 score) indicating room for improvement
- Necessity for rigorous human oversight and testing before deploying patches

Academic Researcher

Amisha Bhaskar

InterPatch: Advanced Vulnerability Patching for Multi-Component Software Systems

Extending LLMPATCH to Address Inter-Component Vulnerabilities

Objectives

- To handle complex dependencies and interactions between different components.
- To improve security in integrated environments like microservices or modular applications.

Core Components of InterPatch

1. Semantic Analysis Expansion through Graph-Based Code Representation

LLMPATCH leverages Program Dependence Graphs (PDGs) to analyze vulnerabilities within a single component by capturing data and control dependencies. For InterPatch, we need to adapt and expand this concept to a System Dependence Graph (SDG) that incorporates multiple components and possibly different technologies.

Mathematical Foundation:

- **PDG**: Given a program consisting of statements a PDG $s_1, s_2, \ldots, s_n,$ is a directed graph G = (V, E), where each vertex $(v_i, v_j) \in E$ corresponds to a statement $s_i,$ and an edge $v_i \in V$ represents a data or control dependency.
- **SDG**: An SDG extends the PDG by incorporating vertices and edges that represent inter-component interactions. If components C_1 and C_2 have interactions based on data or control flows, these are added to the graph, providing a holistic view of the software system's architecture and dependencies.

Approach:

- Develop parsing tools that can construct SDGs by analyzing source code across different programming languages and runtime environments.
- Use these SDGs to identify critical points where vulnerabilities can propagate between components and to analyze the broader impact of potential vulnerabilities within the system.

2. Dynamic Exemplar Generation Using Hybrid Models

The dynamic generation of exemplars in LLMPATCH based on specific vulnerabilities could be improved using hybrid machine learning models that combine supervised learning techniques for vulnerability detection with unsupervised learning for anomaly detection across software components.

Mathematical Foundation:

- **Supervised Learning**: Given a training dataset $D = \{(x_i, y_i)\}_{i=1}^{n}$ where x_i is a feature vector extracted from the SDG and y_i is a label indicating the presence or type of vulnerability, a function $f : X \to Y$ is learned.
- **Unsupervised Learning**: For anomaly detection, clustering techniques like k-means or DBSCAN could be applied to the feature vectors to identify unusual patterns that might suggest vulnerabilities.

Approach:

- Integrate these learning models to continuously update the exemplar database as new patterns and types of vulnerabilities are discovered.
- Use these models to generate context-sensitive prompts for the language models to create more accurate patches.

3. Advanced Patch Validation Techniques

To validate patches in a multi-component environment, we can utilize ensemble methods that combine predictions from multiple models to decide whether a patch is valid.

Mathematical Foundation:

- **Ensemble Methods**: If $\{f_1, f_2, \ldots, f_m\}$ are models that predict the validity of a patch, the final decision can be made based on a majority vote or weighted aggregation of these predictions: $F(x) = \text{argmax}_y \sum_{j=1}^{m} w_j \cdot f_j(x, y)$ where w_j is the weight assigned to the model f_j based on its accuracy.

Approach:

- Deploy multiple language models fine-tuned on different aspects of software systems (e.g., front-end, back-end, database) to evaluate patches.
- Use these models to simulate the application of patches in virtual environments and monitor for functional and security regressions.

Expected Challenges

- **Complex Dependency Resolution**: Understanding and resolving dependencies and interactions between different software components can be significantly more complex than handling single-component vulnerabilities.
- **Diverse Environment Handling**: Dealing with different programming languages, frameworks, and environments within the same software system will require versatile and robust parsing and analysis tools.

Impact

- **Broader Applicability**: This project would expand the applicability of automated vulnerability patching systems to more complex and realistic software environments, such as modern microservices architectures and integrated platforms.
- **Enhanced Security**: By ensuring that inter-component vulnerabilities are effectively patched, the overall security of multi-component software systems can be significantly improved, protecting against more sophisticated attack vectors.

Private Investigator

Andy Lin

Yu Nong

Education:

- BSc in Automation at South China University of Technology
- MS in Computer Science at Washington State University
- Now pursuing PhD in Computer Science at Washington State University

Work Experience

- Front-end engineer to oversee a large-scale digital cooperation platform for national hospitals at Beijing Hantang Technology Stock Company.
- IT helpdesk tech intern to maintain network infrastructures at Intuitive Networks.
- Software testing consultant to examine software quality at Optimum Semiconductor.

Yu Nong

Previous Projects

A Preliminary Study on Open-Source Memory Vulnerability Detectors

-> He benchmarked several static analysis tools to find memory-related vulnerabilities.

Open Science in Software Engineering: A Study on Deep Learning-Based Vulnerability Detection

-> He explored the reproducibility and transparency of deep learning models used for vulnerability detection.

Yu Nong

Motivation

- Stems from the need for effective and timely vulnerability patching due to the rapid increase in cyber threats.
- Hence, his research interests are about applying deep learning models to examine whether a computer program contains vulnerabilities.
- For example. LLMPATCH pushes the boundaries of LLM applications in computer security to address challenges in vulnerability patching through deep learning solutions.

Social Impact Assessor

Sonal Kumar

Positive Social Impacts Self-Assessed in the Paper

Enhanced Cybersecurity through Automated Patching: The paper emphasizes that LLMPATCH can automate vulnerability patching, including zero-day vulnerabilities, which are critical to improving software security. By addressing security flaws more efficiently, it aims to prevent cyberattacks that exploit software vulnerabilities, potentially reducing the frequency and severity of cybersecurity incidents .

Efficiency in Patch Development: LLMPATCH reportedly produces patches more quickly and accurately than traditional methods, suggesting that developers and companies could deploy it to maintain more secure software without extensive manual intervention. This has social benefits in protecting users' data privacy and security at scale.

Reduction in Resource Costs for Vulnerability Management: By automating parts of the patching process, the paper suggests that LLMPATCH can reduce costs associated with manual vulnerability analysis and patching. This cost-saving aspect could make security more accessible to smaller organizations without extensive cybersecurity resources.

Potential Positive Impacts Not Addressed

Increased Accessibility to Security for Smaller Entities: Small and medium-sized enterprises (SMEs), which often lack the resources for advanced cybersecurity, could benefit from using an automated tool like LLMPATCH to manage vulnerabilities without extensive technical expertise.

Potential for Broader Applications of LLM-based Security Tools: While the paper focuses on vulnerability patching, a framework like LLMPATCH could inspire other applications in cybersecurity, such as automated detection and prevention strategies, which would broadly benefit secure software development.

Potential Negative Impacts Overlooked

Risk of Over-Reliance on Automation in Security: The adoption of LLMPATCH could lead organizations to become overly dependent on automated solutions, potentially reducing vigilance and manual scrutiny in security management. Automated systems can fail to capture nuanced vulnerabilities that require human expertise.

Potential Misuse of Automated Patching Technology: While designed for positive use, tools like LLMPATCH could potentially be reverse-engineered or repurposed by malicious entities to identify or even create vulnerabilities in existing software, exacerbating security issues rather than mitigating them.

Economic Implications for Cybersecurity Jobs: Automation in vulnerability patching might reduce the demand for certain cybersecurity roles, impacting employment in sectors that rely on manual patch development and vulnerability management, especially if these tools are widely adopted.

Limitations in Accuracy and False Positives: The paper mentions the potential for hallucinations and inaccuracies in LLM predictions, meaning that incorrect patches could be generated. Deploying such patches without thorough human review could lead to new vulnerabilities or break existing functionality, potentially impacting users and clients reliant on stable software systems.