PatUntrack: Automated Generating Patch Examples for Issue Reports without Tracked Insecure Code

Ziyou Jiang[1,2,3], Lin Shi[4], Guowei Yang[5], Qing Wang[1,2,3*]
[1]State Key Laboratory of Intelligent Game, Beijing, China;
[2]Science and Technology on Integrated Information System Laboratory,
Institute of Software Chinese Academy of Sciences, Beijing, China;
[3]University of Chinese Academy of Sciences, Beijing, China;
[4]School of Software, Beihang University, Beijing, China;
[5]The University of Queensland, Brisbane, Australia
{ziyou2019, wq}@iscas.ac.cn, shilin@buaa.edu.cn, guowei.yang@uq.edu.au

Abstract

Security patches are essential for enhancing the stability and robustness of projects in the open-source software community. While vulnerabilities are officially expected to be patched before being disclosed, patching vulnerabilities is complicated and remains a struggle for many organizations. To patch vulnerabilities, security practitioners typically track vulnerable issue reports (IRs), and analyze their relevant insecure code to generate potential patches. However, the relevant insecure code may not be explicitly specified and practitioners cannot track the insecure code in the repositories, thus limiting their ability to generate patches. In such cases, providing examples of insecure code and the corresponding patches would benefit the security developers to better locate and resolve the actual insecure code. In this paper, we propose PatUntrack, an automated approach to generating patch examples from IRs without tracked insecure code. PatUntrack utilizes auto-prompting to optimize the Large Language Model (LLM) to make it applicable for analyzing the vulnerabilities described in IRs and generating appropriate patch examples. Specifically, it first generates the completed description of the Vulnerability-Triggering Path (VTP) from vulnerable IRs. Then, it corrects potential hallucinations in the VTP description with external golden knowledge. Finally, it generates Top-K pairs of *Insecure Code and Patch Example* based on the corrected VTP description. To evaluate the performance of PatUntrack, we conducted experiments on 5,465 vulnerable IRs. The experimental results show that PatUntrack can obtain the highest performance and improve the traditional LLM baselines by +17.7% (MatchFix) and +14.6% (Fix@10) on average in patch example generation. Furthermore, PatUntrack was applied to generate patch examples for 76 newly disclosed vulnerable IRs. 27 out of 37 replies from the authors of these IRs confirmed the usefulness of the patch examples generated by PatUntrack, indicating that they can benefit from these examples for patching the vulnerabilities.

[*]Corresponding author.

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3694982

ACM Reference Format:
Ziyou Jiang[1,2,3], Lin Shi[4], Guowei Yang[5], Qing Wang[1,2,3]. 2024. PatUntrack: Automated Generating Patch Examples for Issue Reports without Tracked Insecure Code. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3694982

1 Introduction

Security patches are essential for enhancing the stability and robustness of projects in the Open-Source Software (OSS) community. In 2017, the Software Engineering Institute (SEI) at Carnegie Mellon University released the CERT Guide to Coordinated Vulnerability Disclosure (CVD) [10], which officially states that individuals or organizations should "*deploy a patch or take other remediation action*" before they disclose a vulnerability to the public security databases [3, 33], such as Common Vulnerabilities and Exposure (CVE) [6]. However, patching vulnerabilities is complicated and remains a struggle for many organizations [56]. For example, a vulnerability in the project *python-markdown2* has "*no fix*" and "*welcome pull requests*" for a long time, as is reported in the issue *trentm/python-markdown2/issues/285* [4]. These unpatched vulnerabilities can be utilized by attackers through deploying exploits to harm the affected systems (e.g., zero-day [17] and one-day [23, 26] attacks), resulting in millions dollars of business losses [63].

OSS developers typically report vulnerabilities through the issue reports (IRs) [78]. Security practitioners, who manage the vulnerability disclosure, can then track these IRs with issue-tracking systems [5, 16], and analyze the relevant insecure code to generate potential patches. However, the relevant insecure code may not be explicitly specified and practitioners cannot track the insecure code in the repositories, thus limiting their ability to generate patches. In such cases, providing examples of insecure code and the corresponding patches would benefit the security developers to better locate and patch the actual insecure code. In general, generating example insecure code and patches is challenging, mainly due to the semantic gaps between code and natural language, as well as the information omission in developer's description of the venerability in the IRs. Our preliminary study in Section 2.1 shows that 69.0% of vulnerable IRs have no insecure code tracked with either manual analysis or State-of-the-Art (SOTA) commit trackers [68, 93], and over 70% of them were successfully exploited by attackers. These results inspire us to design an automated approach to generate patch examples based on the IRs without tracked insecure code, which can help security practitioners patch vulnerabilities soon after the IRs are created by the developers.

In this paper, we propose an automatic approach, i.e., PatUntrack, which generates patch examples from IRs without tracked insecure code. It optimizes the Large Language Model (LLM) with auto-prompting [73] to make it applicable for analyzing the types and triggering logic of vulnerabilities from their textual descriptions, and generating appropriate patches. PatUntrack consists of three main steps: ❶ First, it generates the description of the Vulnerability Triggering Path (VTP) from IR, which captures how the vulnerability is triggered. ❷ Second, it corrects potential hallucinations in the VTP description with external golden knowledge. ❸ Third, it utilizes the VTP description to predict the patch types and generates Top-K pairs *Insecure Code and Patch Example*.

To evaluate the performance of PatUntrack, we conducted experiments on 5,465 vulnerable IRs. The results show that PatUntrack achieves the highest performance and improves the traditional LLM baselines by +17.7% (MatchFix) and +14.6% (Fix@10) on average in patch example generation. Furthermore, we applied PatUntrack to generate patch examples for 76 newly disclosed vulnerable IRs that do not have tracked insecure code, and asked the authors whether our patch examples can assist them with patching the vulnerabilities. We have received replies from the authors for 37 IRs, and 27 replies confirmed the usefulness of the patch examples generated by PatUntrack, indicating that they can benefit from these examples for patching the vulnerabilities. To summarize, this paper makes the following main contributions:

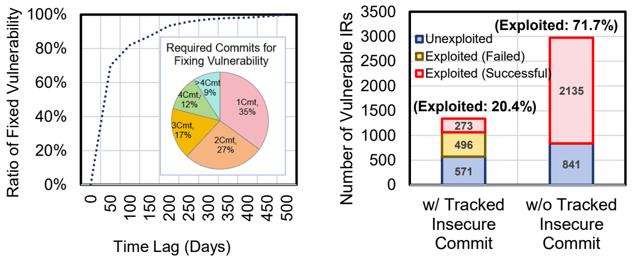
- **Technique**: PatUntrack, an automated approach to generate patch examples for vulnerable IRs without tracked insecure code. To the best of our knowledge, this is the first work on generating patch examples without guidance from the source code.
- **Evaluation**: An experimental evaluation of PatUntrack that shows that PatUntrack outperforms all baselines on generating insecure code & patch examples, as well as a human evaluation on newly-disclosed vulnerable IRs that further demonstrates its usefulness in practice.
- **Data**: The datasets and source code [9], which are made publicly available to facilitate the replication and the application of PatUntrack in the more extensive contexts.

2 Preliminaries

In this section, we first conduct the preliminary study by analyzing the time cost of raising patches for the vulnerable IRs and the exploited ratio of IRs with/without tracked insecure commits. Then, we provide an example to illustrate the motivation of PatUntrack.

2.1 Preliminary Study of Vulnerability Patching

To analyze the time cost of vulnerability patching, we introduce a widely-used vulnerability dataset, i.e., GHArchive [8], which achieves the IRs in the software community. Among them, GHArchive contains vulnerable IRs with their original information of CVE-disclosed vulnerabilities, such as IR's textual descriptions, commits for insecure code and patch (indicated by links with string `TCommit` and `TPatch`), etc. We first analyze the time lag between the IR creation and the vulnerability patching by referring to the commit links to the vulnerable IRs containing insecure commits. Figure 1 (a) shows that nearly 20% of the vulnerable IRs require over 150 days to raise the patches for successfully fixing the vulnerabilities, and 38% require over 3 commits to fix the vulnerabilities.



(a) Time cost of fixing vulnerabilities in vulnerable IRs in CVE.

(b) Exploit of IRs with/without tracked insecure commits in CVE.

Figure 1: The preliminary study to analyze the time cost of patching and the exploited ratio of the vulnerable IRs.

Moreover, we also analyze whether the tracked insecure commits may affect vulnerability exploitation by calculating the intersections of vulnerable IRs with tracked insecure commits and exploited vulnerabilities. We first decide whether the vulnerable IR contains the insecure commits with the following steps: ❶ we analyze the commit links in GHArchive; ❷ we utilize the SOTA commit tracker [93] to track the commits if GHArchive does not present the link; and ❸ we manually track the commit links if the previous steps cannot find the commit links.

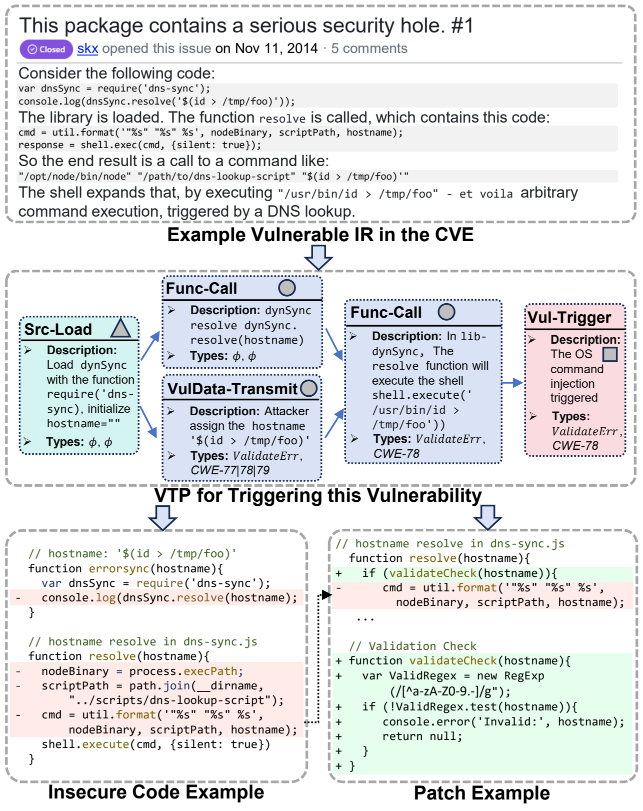
Second, we determine whether the vulnerability is exploited by analyzing the logs in the links with `TExploited` for vulnerability exploitation, e.g., exploited time, IP address, and vulnerable version. We find that there are three types of exploitation as follows:

- **Unexploited:** The vulnerabilities are not exploited by attackers.
- **Exploited (Failed):** The vulnerabilities are exploited after they are fixed with specific patches.
- **Exploited (Successful):** The vulnerabilities are exploited before they are fixed, which means the attackers may harm the systems.

Figure 1 (b) shows that 2,976 of 4,316 vulnerable IRs (69.0%) cannot track the insecure commits from the GHArchive, Among these IRs without tracked insecure code, 71.7% of vulnerabilities in the vulnerable IRs are successfully exploited by attackers, which is +50.3% higher than IRs with insecure commits. These results show that the insecure commits with patches are important to reduce the exploitation of vulnerable IRs, but raising appropriate patches to fix the vulnerabilities is a time-consuming task.

2.2 Motivating Example

Figure 2 shows the motivation of generating insecure code & patch examples from the vulnerable IR. From the example, we analyze how the vulnerability is triggered based on the textual description of IR. The project first loads the library `dynSync` and initializes the parameter `hostname`. Then, it calls the function `dynSync.resolve`, which will execute the system command to look up the DNS server to resolve the `hostname`. However, this function-calling process has vulnerabilities, since the logic of `resolve` function contains the execution of the system command. If the attackers initialize the `hostname` with specific strings like `"$(id > /tmp/foo)"`, it may inject the vulnerabilities and harm the system.


Closed
Example Vulnerable IR in the CVE
Src-Load
Description: Load dynSync with the function require('dns-sync), initialize hostname=""
Types: ϕ, ϕ
Func-Call
Description: dynSync resolve dynSync. resolve(hostname)
Types: ϕ, ϕ
VulData-Transmit
Description: Attacker assign the hostname '$(id > /tmp/foo)'
Types: *ValidateErr*, *CWE-77|78|79*
Func-Call
Description: In lib-dynSync, The resolve function will execute the shell shell.execute(' /usr/bin/id > /tmp/foo'))
Types: *ValidateErr*, *CWE-78*
Vul-Trigger
Description: The OS command injection triggered
Types: *ValidateErr*, *CWE-78*
VTP for Triggering this Vulnerability
Insecure Code Example
Patch Example

This package contains a serious security hole. #1

skx opened this issue on Nov 11, 2014 · 5 comments

Consider the following code:

```
var dnsSync = require('dns-sync');
console.log(dnsSync.resolve('$(id > /tmp/foo)'));
```

The library is loaded. The function `resolve` is called, which contains this code:

```
cmd = util.format('"%s" "%s" %s', nodeBinary, scriptPath, hostname);
response = shell.exec(cmd, {silent: true});
```

So the end result is a call to a command like:

```
"/opt/node/bin/node" "/path/to/dns-lookup-script" "$(id > /tmp/foo)'"
```

The shell expands that, by executing `"/usr/bin/id > /tmp/foo" - et voila` arbitrary command execution, triggered by a DNS lookup.

```
  // hostname: '$(id > /tmp/foo)'
  function errorsync(hostname){
    var dnsSync = require('dns-sync');
-   console.log(dnsSync.resolve(hostname);
  }

  // hostname resolve in dns-sync.js
  function resolve(hostname){
-   nodeBinary = process.execPath;
-   scriptPath = path.join(__dirname,
        "../scripts/dns-lookup-script");
-   cmd = util.format('"%s" "%s" %s',
        nodeBinary, scriptPath, hostname);
    shell.execute(cmd, {silent: true})
  }
```

```
// hostname resolve in dns-sync.js
  function resolve(hostname){
+   if (validateCheck(hostname)){
-      cmd = util.format('"%s" "%s" %s',
         nodeBinary, scriptPath, hostname);
   ...

  // Validation Check
+ function validateCheck(hostname){
+   var ValidRegex = new RegExp
        (/[^a-zA-Z0-9.-]/g");
+   if (!ValidRegex.test(hostname)){
+      console.error('Invalid:', hostname);
+      return null;
+   }
+ }
```

Figure 2: The vulnerable IR and the generated insecure code & patch example (*skoranga/node-dns-sync/issues/1* [1]).

Referring to the secure coding practices in OWASP [61], this vulnerability is a typical OS command injection (CWE-78) [54] and the type of patching is validating the input with *Regex Testing*, so the patch example will incorporate the validation of the inputs. The system commands will not execute if the input strings contain such specific strings, thus preventing the vulnerabilities.

3 Approach

The overall framework of PatUntrack is illustrated in Figure 3. Since the IR authors may miss some details in describing vulnerabilities, we first extract the VTP descriptions from IR's textual description and complete the missing nodes and edges. Second, since the pre-trained data and training strategy of LLMs have flaws that result in hallucinations, we propose VulCoK to correct the hallucinations in VTP descriptions. Third, we jointly generate insecure code & patch examples with patch type prediction with the corrected VTP description. For each step, we utilize auto-prompting to optimize LLM and make it applicable to analyze vulnerabilities.

3.1 Generating Complete VTP Description

The IR can be formulated as follows: $\{Title, Body\}$, where $Title$ is the summarization of the main topic; $Body$ contains a set of sentences that describe the details of IR. PatUntrack generates the complete VTP description by ❶ extracting the original VTP description from the IR textual descriptions, and ❷ completing the missing nodes and edges to update the vulnerable IRs.

3.1.1 *VTP Description Extraction.*

Previously, Cheng et al.'s [19] define a Bug-Triggering Path (BTP) as a set of program statements to reside in the execution paths toward the location where the error is triggered, which is an effective method to detect and reproduce the vulnerabilities, and has been utilized by different vulnerability detectors [43, 44, 55]. However, there are gaps between normal bugs and vulnerabilities, so traditional BTPs are not useful to accurately find vulnerabilities. For example, the traditional BTP defines the operations of the program as *package loading*, *variable declaration*, and *function calling*, which are operations to trigger normal bugs in the OSS projects. On the contrary, vulnerabilities focus on the transmission of tainted data [20], where we find the "*source*" and "*sink*" code to describe the transmission path of the tainted data and locate the code lines that may produce the vulnerabilities.

The VTP description in our work can identify the triggering paths of vulnerability by incorporating the transmission of tainted data. The structure can be modeled as $\mathcal{G}_{VTP} = \langle \mathcal{V}_{VTP}, \mathcal{E}_{VTP} \rangle$, where the $\mathcal{V}_{VTP} = \{Op_0, Op_1, ..., Op_T\}$ is a series of nodes that describe the operations that may result in the vulnerability. The Op_0 is the start operation, which contains the operation to load the sources, such as *loading third-party library* and *initializing variables*, and Op_T is the end operation that indicates the vulnerability is triggered after the previous operations are conducted.

Based on the previous works [19] and our manual analysis on over 1K vulnerabilities, we summarize four types of operations that cover the triggering process of vulnerabilities. we manually analyzed the IRs with experienced security practitioners who participated in our data annotation to determine the types of operation nodes/edges with **Open Card Sorting** [69], a flexible classification method that allows us to create information categories freely, thus helping designers develop more appropriate types.

- **Src-Load:** This operation indicates that the program loads the vulnerability-related source data, such as loading the packages that may have vulnerabilities and loading the variables that may contain the taint information.
- **Func-Call:** Since some vulnerabilities are directly caused by the incorrect calling of the functions, such as the use-after-free [87] vulnerability, we specifically analyze the function calling processes that may trigger the vulnerabilities.
- **VulData-Transmit:** This operation denotes the transmission of the vulnerable data. The data comes from the source variables or the different libraries, and will finally transmit to the sink code lines that may harm the system.
- **SecData-Transmit:** This operation indicates the transmission of other data in the function calling process. Different from the VulData-Transmit, these data are secure and will not harm the system, and we also analyze the transmission of secure data to distinguish it from vulnerable data.
- **Vul-Trigger:** We add this node to intuitively indicate the results of vulnerability triggering.

The edges in VTP description $(Op_i \rightarrow Op_j) \in \mathcal{E}_{VTP}$ is the one-way link that denotes the transition of how to trigger the vulnerability, where Op_i is the prerequisite operation for the Op_j. With the extracted VTP description, we can generate the insecure coding example which can reflect the vulnerable code in the projects, and generate patches based on the coding examples.

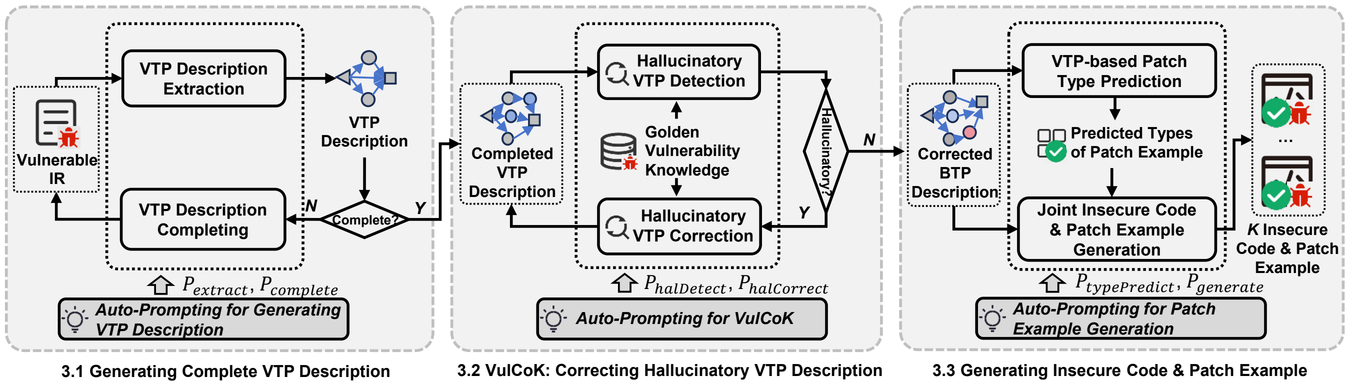

Vulnerable IR
VTP Description Extraction
VTP Description
Complete?
N
Y
VTP Description Completing
$P_{extract}, P_{complete}$
Auto-Prompting for Generating VTP Description
3.1 Generating Complete VTP Description
Completed VTP Description
Hallucinatory VTP Detection
Golden Vulnerability Knowledge
Hallucinatory VTP Correction
Hallucinatory?
$P_{halDetect}, P_{halCorrect}$
Auto-Prompting for VulCoK
3.2 VulCoK: Correcting Hallucinatory VTP Description
Corrected BTP Description
VTP-based Patch Type Prediction
Predicted Types of Patch Example
Joint Insecure Code & Patch Example Generation
K Insecure Code & Patch Example
$P_{typePredict}, P_{generate}$
Auto-Prompting for Patch Example Generation
3.3 Generating Insecure Code & Patch Example

Figure 3: The structure of PatUntrack.

Each node Op_i is a triplet $\langle Op\_Type, Op\_Desc, Vul\_Type \rangle$, where ❶ The element Op_Type is the previous type of operation node. ❷ The Op_Desc is the description of the operations, which briefly explains the information of each operation step with texts and a few code snippets. ❸ The Vul_Type is the type of vulnerability, and we introduce the *CWE type* and the detailed *error types* in it to describe the vulnerability. Chow et al. [21] indicates that for different error types, the **focuses** of bug triggering methods are different. Inspired by it, we define seven error types in VTP based on our manual analysis of vulnerabilities, as shown in Table 1.

Table 1: The error types in the VTB operation nodes.

ID	Error Types (VTP)	Description
1	Encoding & Validation Errors	The error comes from improperly handled, leading to unexpected behavior.
2	Dependency Errors	The variables, functions, or libraries are unresolved or incorrect.
3	Injection & Logic Errors	The functions are improperly used leading to incorrect execution.
4	Memory Management & Concurrency Errors	The way how data is handled has errors, leading to exploitable conditions.
5	Race & Configuration Errors	The error relates to the system conditions, affecting its correct operation.
6	Buffer overflow	The error involves improper handling of boundaries and limits in storage, leading to overflows.
7	Error handling & logging issues	The encoding or decoding have errors, leading to XSS or injection attacks.

Based on the above definitions, we format the structure of the prompts P_0 in the PatUntrack, as is shown in Figure 4. Each prompt consists of the following three components: ❶ **Task Definition**, where we define the problem that LLM should resolve; ❷ **Details of VTP Description**, where we incorporate the definition of the VTP operation, i.e., operation type, descriptions, and vulnerability type Vul_Type, as well as the transmissions between edges. The Vul_Type contains two parts of the types, i.e., the error and CWE types, where Common Weakness Enumeration (CWE) [7] is a more specific categorization that differentiates the types of vulnerabilities with their causes and behaviors; and ❸ **Focus-List**, where we provide some focuses that guide LLM to resolve the problem. The focus list contains a set of pairs $\langle Vul\_Type_i, f_i \rangle$, where f_i is the focus of Vul_Type when generating VTP descriptions. We introduce the focus list due to the following two reasons:

(1) Some historical IRs may contain guidance on how to generate patch examples. LLMs can refer to this information to guide the generation of VTP descriptions and reduce output biases.
(2) Some IRs may lack detailed information to introduce the triggering process of vulnerability, so LLMs cannot directly generate the VTP description based on the contents. By referring to the focus list, LLMs can utilize historical information to complete the missing information in these IRs.

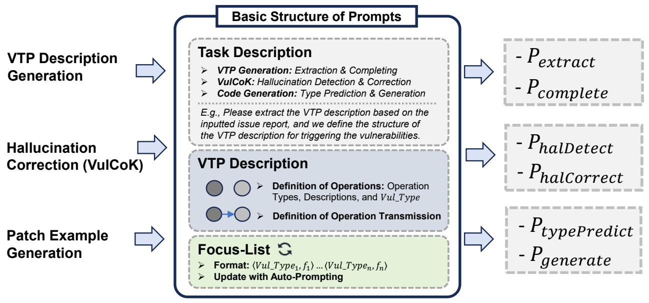

VTP Description Generation
Hallucination Correction (VulCoK)
Patch Example Generation
Basic Structure of Prompts
Task Description
VTP Generation: Extraction & Completing
VulCoK: Hallucination Detection & Correction
Code Generation: Type Prediction & Generation
E.g., Please extract the VTP description based on the inputted issue report, and we define the structure of the VTP description for triggering the vulnerabilities.
VTP Description
Definition of Operations: Operation Types, Descriptions, and Vul_Type
Definition of Operation Transmission
Focus-List
Format: $\langle Vul\_Type_1, f_1 \rangle \dots \langle Vul\_Type_n, f_n \rangle$
Update with Auto-Prompting
- $P_{extract}$
- $P_{complete}$
- $P_{halDetect}$
- $P_{halCorrect}$
- $P_{typePredict}$
- $P_{generate}$

Figure 4: The prompt format P_0 in the PatUntrack.

The f_i will be updated when LLM analyzes the historical IRs with the auto-prompting, and we will discuss the auto-prompting in the following sections in detail. To extract the VTP description, we will design the prompt $P_{extract} \leftarrow P_0$ by replacing the task description in this prompt format P_0 with instructions for VTP extraction.

3.1.2 VTP Description Completing. Since some VTP descriptions are not complete and miss some essential operations and transitions in the paths, PatUntrack will first detect whether the extracted VTP description is complete. Then, it will complete the missing operation nodes and transitions. PatUntrack detects the *operation-level* and *transition-level* completeness, and adds the missing information in the nodes and edges in these levels.

- **Operation-level Completeness:** PatUntrack detects whether the VTP description misses some intermediate operation nodes Op_{miss}, or the existing operation node Op_i misses some information, such as the description of insecure code and Vul_Type misses the error or CWE types. PatUntrack will ask the LLMs to reason the missing information within the operation nodes, or generate some new intermediate nodes between the existing operation nodes to complement the missing flows.
- **Transition-level Completeness:** The transitions between the operations $Op_i \rightarrow Op_j$ are missed, so the logic flow is not complete to trigger the vulnerability, and PatUntrack will add these transition edges to complement the VTP description.

Algorithm 1: Process of Type-based Auto-Prompting.

```
Input: The original prompt $P_0$, the dataset with labeled insecure
       codes and patches $LabeledDataset$, the predicted type
       $Vul\_Type$, and the score function for specific task $F_s$.
Output: The generated prompt $P_T$.
1  $P_T \leftarrow P_0, Upd\_Prompts \leftarrow \{INSERT|DELETE|MODIFY\}$;
2  Function Auto-Prompting($P_0$, $F_s$):
3      $f = Focus[Vul\_Type]$;
4      for $item \in LabeledDataset$ do
5          $s_T = F_s(item.truth, item.pred, P_T)$;
6          $\{P_{insert}.f, P_{delete}.f, P_{modify}.f\} = LLM(P_T.f, Upd\_Prompts)$;
7          $P_T \leftarrow \{P_{insert}|P_{delete}|P_{update}\}$;
8          $P_T = \arg\max_{P_T}(s_T - F_s(item.truth, item.pred, P_T))$;
9      end
10 end
11 return $P_T$;
```

The prompt for VTP description $P_{complete}$ completion utilizes the same format P_0. The only difference is that we add the definition of the previous completeness. The prompt will also contain the different focus f_i for different vulnerability types.

3.1.3 Auto-Prompting for Generating VTP Description. Since some of the LLMs cannot be directly fine-tuned, such as ChatGPT, the researchers have utilized the text generation ability of LLM to design the specific prompt for each input, i.e., Auto-Prompting [73]. We design a meta-framework for auto-prompting, as is shown in Algorithm 1. It utilizes the score function F_s to calculate the differences between predicted and ground-truth labels in the labeled dataset and updates the f_i in the prompt. The line 1 indicates that the auto-prompting updates the prompt's focus f_i by *inserting*, *deleting*, and *modifying* elements in the original prompts. The auto-prompting process controls the prompt updating with simple prompts, such as "*Please update the prompt by inserting|deleting|modifying the [item] to the prompt's focus f*", where *[item]* is the sample used to optimize the prompt. The samples come from the historical IRs that can track the ground-truth insecure code and patch. The line 3~8 indicate that PatUntrack utilizes a score function F_s to analyze the similarity between LLM outputs and ground-truth (*lower the score, higher the similarity*). We select the most appropriate prompt P_T based on the score differences among these three updated prompts.

The score function of auto-prompting the VTP description extractor and completer is calculated by analyzing the *matching and masking scores*, which can be formulated as follows:

$$F_s(VTP\_Code^-|VTP\_[M], VTP\_IR|VTP\_[M]', P_{extract}|P_{complete}) = \underbrace{sim(VTP\_Code^-, VTP\_IR)}_{score_{match}} + \underbrace{sim(VTP\_[M], VTP\_[M]')}_{score_{mask}} \quad (1)$$

where F_s is the score function that calculates the sum of two scores, i.e., $score_{match}$ and $score_{mask}$. The first score analyzes whether LLM can accurately generate the VTP descriptions that reflect the triggering process of vulnerabilities, and $socre_{mask}$ analyzes whether LLM can complement the incomplete IRs.

For $score_{match}$, The VTP_IR is the generated IRs with original prompt $P_{extract}$, and VTP_Code^- is the ground-truth triggering path from the insecure code. We utilize the edit distance, i.e., Levenshtein Distance [15], as the similarity, which is useful to measure the similarity between two texts. For $score_{mask}$, we randomly select some nodes in the extracted VTP, then reflect them to the original IR and mask these chosen texts $VTP\_[M]$. We utilize the LLM to predict the masked text to $VTP\_[M]'$ and calculate the edit distances between them. We utilize these scores to measure the performance of LLM on generating VTP descriptions and update the prompts with the meta-framework.

Table 2: The golden external dataset $\mathcal{D}$ of VulCoK.

Id	Golden Dataset	Last Updated	Link
1	SARD	2024	https://samate.nist.gov/SARD/
2	OWASP	2024	https://owasp.org/www-project-benchmark
3	Debian	2024	https://bit.ly/3bX30ai
4	VDISC	2024	https://osf.io/d45bw/

3.2 Correcting Hallucinatory VTP Description

In Huang et al's survey [34], they indicate that the pre-trained data, training, and decoding strategies of LLMs have flaws that result in content that is inconsistent with real-world facts, which is called LLM hallucinations. To address the hallucinations, Li et al. [42] proposed the CoK, which utilizes external golden knowledge for the hallucination correction. Inspired by this work, we propose the VulCoK, as is shown in Figure 5, which can correct the hallucinations in VTP operation nodes and transition edges.

3.2.1 Hallucinatory VTP Detection. The detection process of VTP description contains two parts. i.e., Vul-Type Hallucination Detection and Description Hallucination Detection, which detects the hallucinations in vulnerability types and descriptions of VTP nodes. We first introduce the external golden databases Table 2, which is selected based on the update time, the usage of databases in industry and research, and the number of vulnerabilities. We detect the hallucinations in the VTP description with the Breadth-First Search (BFS) [76], which searches for the current operation item $OpItem$ and its connected operations $\{OpConn|OpItem \rightarrow OpConn\}$. The LLM first generates the queries for retrieving the golden items in the dataset $\mathcal{D}$. Similar to Section 3.1.2, we also utilize the operation and the historical transitions to analyze whether they contain the hallucination. We utilize the prompt $P_{halDetect}$ with the prompt format P_0 to detect the hallucination, which contains the definition of hallucinations, as well as the focus list of CWE and error types.

3.2.2 Hallucinatory VTP Correction. For the VTP descriptions that contain the hallucinations, i.e., the hallucinations from type and description in VTP. First, suppose the CWE and error types are incorrect and contain hallucinations. In that case, the VulCoK needs to correct the hallucinatory types in the node $OpItem$ and $OpConn$. If the types are correct, the LLM needs to correct the hallucinations of the VTP description and $OpNext$ and re-generate the transitions for the new VTP operations. After these corrections, the original $\mathcal{G}_{VTP}$ will be updated to $\mathcal{G}_{VTP}^{'}$, and the current item $OpItem$ will move to its connected items $OpConn$. The prompt $P_{halCorrect}$ utilizes the same format P_0 for correcting the hallucinations. It directly asks LLM to correct the VTP's vulnerability types and descriptions based on the retrieved golden knowledge, and it will also incorporate the focus list of different vulnerability types.

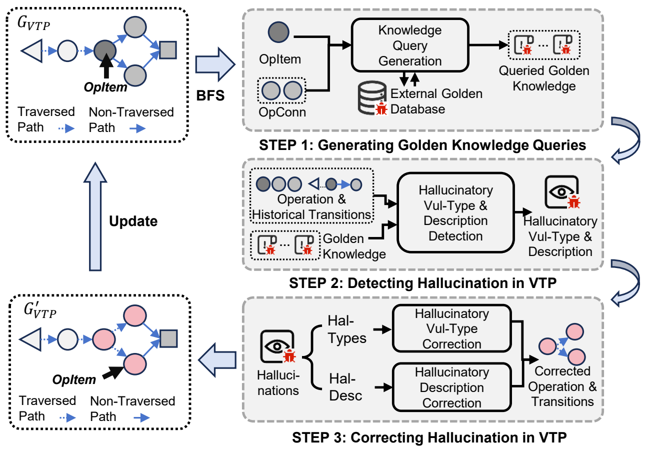

$\mathcal{G}_{VTP}$
OpItem
Traversed Path
Non-Traversed Path
BFS
OpItem
OpConn
Knowledge Query Generation
External Golden Database
Queried Golden Knowledge
STEP 1: Generating Golden Knowledge Queries
Update
Operation & Historical Transitions
Golden Knowledge
Hallucinatory Vul-Type & Description Detection
Hallucinatory Vul-Type & Description
STEP 2: Detecting Hallucination in VTP
$\mathcal{G}'_{VTP}$
OpItem
Traversed Path
Non-Traversed Path
Hallucinations
Hal-Types
Hal-Desc
Hallucinatory Vul-Type Correction
Hallucinatory Description Correction
Corrected Operation & Transitions
STEP 3: Correcting Hallucination in VTP

Figure 5: The logic flow of VulCoK.

3.2.3 Auto-Prompting for VulCoK. Since the retrieved knowledge in the selected golden dataset incorporated the insecure code, the score function of auto-prompting the VulCoK is calculated by analyzing the similarity between the tracked insecure code and the golden knowledge's insecure code.

$$F_s(CodeGold^-, Code^-, P_{halDetect}|P_{halCorrect}) = \sum_{ColdGold^-} sim(CodeGold^-, Code^-) \tag{2}$$

where the $CodeGold^-$ indicates the insecure code in the golden knowledge, and the sim is the edit distance. We sum all the distances in the retrieved knowledge, then we feed the F_s into Algorithm 1 and update the prompt with this meta-framework.

3.3 Generating Insecure Code & Patch Example

In this section, PatUntrack first predicts the patch types based on the corrected VTP description. Then, it jointly generates the insecure coding & patch examples based on the patch types.

3.3.1 VTP-based Patch Type Prediction. Previously, Chow et al. [21] defines 12 patch types for fixing the normal bugs in OSS projects. They construct a mapping from the error types to the patch types, which reflect which types of patches are more frequently used for fixing certain bugs. Inspired by this work, we also ask the LLMs to predict the patch types before the patch example generation. First, since the manual investigation of vulnerable IRs with patches shows the types of patches for fixing the vulnerabilities are similar to fixing the normal bugs, we directly migrate the patch type defined by Chow et al. to our patch type prediction. Then, we ask the LLM to predict the patch type $Patch\_Type$ for the VTP description $\mathcal{G}_{VTP}$ and record the co-occurrence between the vulnerability type and patch type in the current predicted IRs $freq = \#(Vul\_Type, Patch\_Type)/\#Predicted\_IR$. The prompt $P_{typePredict}$ incorporates the definition of patch types and the frequency, as well as the type and focus list in CWE and error types.

3.3.2 Joint Insecure Code & Patch Example Generation. After the prediction of patch types, we generate the patch example based on the $Patch\_Type$ and the original VTP description $\mathcal{G}_{VTP}$. Since some nodes are operations in the imported third-party libraries, we first ask the LLMs to select the nodes and edges that reflect the developer's insecure coding process. Then, we utilize the selected nodes and edges to jointly generate the pairs with insecure coding and patch examples. The idea of joint generation comes from multitask learning [41], where the incorrect output elements are modified based on other output elements' results, thus improving the accuracy of patch generation. The prompt $P_{generate}$ asks LLMs to select nodes/edges and jointly generate the pairs, as well as incorporates the focus list of CWE and error types

3.3.3 Auto-Prompting for Patch Example Generation. The auto-prompting process for the patch example generation utilizes the edit similarity between generated code and ground-truth code in the historical IRs to optimize the prompts, and the score function is shown as follows:

$$F_s(Code^-|Patch^+, Code'|Patch', P_{typePredict}|P_{generate}) = sim(Code', Code^-) + sim(Patch', Patch^+) \tag{3}$$

where the $Code'$ and $Patch'$ are the generated insecure coding and patch examples. The $Code^-$ and $Patch^+$ are the ground-truth of insecure code and patch of the vulnerabilities. We also feed the F_s into Algorithm 1 and update the prompt with the meta-framework.

4 Experimental Design

To evaluate the performance of PatUntrack, we will investigate the following three research questions (RQs)

- **RQ1: How does PatUntrack perform on generating insecure code examples?**
- **RQ2: How does PatUntrack perform on generating patch examples for fix the vulnerability?**
- **RQ3: How does PatUntrack handle IRs when they lack detailed information?**
- **RQ4: How does each component contributes to the PatUntrack on generating insecure coding and patch examples?**

4.1 Dataset Preparation

In this section, we first enrich the IRs from original **GHArchive** [8] with other two representative data sources, i.e., **D2A** [94] and **PatchDB** [80]; then, we denoise the D2A dataset to improve its quality; third, we preprocess the dataset with token replacement and split the dataset into IRs for auto-prompting and evaluation.

STEP 1: Collecting the Dataset. We collect the dataset from three major sources following previous works [39, 64]. The first data source is **GHArchive** [8], a comprehensive dataset that contains over 120K GitHub IRs since 2015. The second data source is **D2A** [94], which is built from real-world vulnerability prediction scenarios and contains over 10K insecure code found from GitHub IRs with their vulnerability types. The third data source is **PatchDB** [80], which incorporates over 4K security patches in the GitHub repositories. All the datasets have been widely used in multiple vulnerability identification tasks [18, 44, 46, 57, 64, 70]. We collect the vulnerability information in these two data sources by searching commit messages and vulnerable IRs, then we remove items without the searched vulnerable IRs.

STEP 2: Denoising the Dataset. The D2A is automatically built by the commit message analyzer, and the authors report that D2A only has 53% accuracy in extracting commits. Therefore, we remove 67

Table 3: The size of the dataset with three sources.

Data Sources	#Vul-IRs	#Auto-Prompt	#Evaluation w/ Labels	#Evaluation w/o Labels
GHArchive	4,316	1,072	268	2,976
D2A	662	530	132	-
PatchDB	487	390	97	-
Total	*5,465*	*1,992*	*497*	*2,976*

noisy samples from D2A as follows: ❶ we obtain the commit messages, and vulnerable IRs by manually searching the repositories; ❷ we remove the noisy items that the commit messages and IRs explicitly indicate that they do not contain the vulnerabilities; and ❸ in the remaining code, we remove the noisy items by checking whether the disclosed vulnerabilities are depreciated in the CVE. To reduce the biases in the data-denoising process, we have invited three security practitioners with over 5-year experience to determine whether the dataset is correctly denoised. We ask them to independently check whether the removed noisy items are accurate. The average Cohen's Kappa [65] value is over 0.9, which means they highly agree on the noisy data removal.

STEP 3: Preprocessing the Dataset. The GitHub IRs collected from the web pages are in XML format, and we need to preprocess the IRs by ragging screenshots and code snippets. We preprocess the IRs with the following procedures: ❶ we first utilize the Tencent OCR to transit the screenshots (wrapped by XML tag `<a href=".jpg|.png">`) to the text [77], then use [SCR] to tag the screenshots, and [CODE] to tag the code snippets (wrapped by XML tags `<code>`, `</code>`). The content will be [CODE] {content of code snippet} after tagging; ❷ we merge similar code snippets and page screenshots, which may have few differences and describe similar vulnerability information; and ❸ following the previous works [72], we remove other XML tags and retain the plain text inside, then we correct typos with Spacy [25]. We utilize 80% of the IRs with code commits for auto-prompting the LLMs, and the rest of the IRs for evaluation. In consequence, the evaluation dataset contains IRs with/without code labels.

Table 3 shows the number of vulnerable IRs for auto-prompting and evaluation. In total, we have obtained 5,465 vulnerable IRs from these three sources, where 1,992 for auto-prompting and 3,473 for evaluating the PatUntrack. Among the evaluation IRs, 2,976 IRs do not contain the tracked insecure code & patches (**w/o Labels**).

4.2 Experimental Baselines

Non-LLM Baselines for Code Generation. CodeBert [27] is a large code model pre-trained on millions of code snippets with the BERT model. We fine-tune the CodeBert on the dataset for auto-prompting. **Codeium** [11] is a low-cost AI-driven approach for code completion and searching. We utilize these baselines to generate insecure code examples from the description of IR. Compared with other baselines, they achieve SOTA performances in our task.

Non-LLM Baselines for APR. The APR tools also utilize the fine-tuned **CodeBert** as the baseline. **InCoder** [28] is designed for code infilling by adopting a causal masking objective. We fine-tune these two models on the ⟨*Insecure Code, Patch*⟩ pairs of evaluation dataset for auto-prompting. To keep these APR baselines consistent with PatUntrack, these two models generate the patches based on the generated insecure code example of PatUntrack. Compared with other baselines, they achieve SOTA performances in our task.

Baselines with Generative LLMs. Recently, researchers have utilized the LLMs with prompt learning to automatically generate code and repair the bugs. We choose the three common LLM baselines in our tasks, which can achieve the SOTA performances. **CodeT5** [81] is pre-trained on T5, which is an encoder-decoder model that takes into account token type information in the code. **Codex (GPT-3)** [91] and **ChatGPT (i.e., GPT-3.5)** [58] are two novel LLMs proposed by OpenAI, which use over 100B of parameters and are trained on over 10TB samples with multiple training strategies (few-shot, zero-shot, etc.). We choose the stable and well-maintained versions: *t5-base* [12], *text-davinci-003* [60], and *gpt-3.5-turbo* [59], and use the ***same prompt in Section 3.3***. Except for the ChatGPT, all the baselines are ***fine-tuned*** on our dataset, then generate code examples and predict types.

4.3 Metrics and Experimental Settings

Metrics. The first metric is the **MatchLine**, which is a strict metric that measures the proportion of total matched statements with the ground-truth code. The second is the **MatchTrig** and **MatchFix**, which measure the matching rate of statements that may contain insecure code (annotated with "-") and patch (annotated with "+"). These two metrics indicate whether the generated code can trigger or fix the vulnerabilities. We choose the $K = 10$ as the default value to measure these matching rates. **AccType** is utilized to measure the accuracy of type prediction, and it measures the average of both CWE and error types in insecure code examples. We also use the **Triggering Rate** (**Trig@**K) and **Fixing Rate** (**Fix@**K) to measure the triggering and fixing rate of generated code examples:

$$Trig@K = \frac{\#Trig\_Vul@K}{\#Total\_Vul@K}, Fix@K = \frac{\#(Trig\_Vul@K \cap Fix\_Vul@K)}{\#Total\_Vul@K} \quad (4)$$

where "#" is the symbol of the number calculation of evaluation samples, and Fix@$K = 1$ if both the vulnerability triggering and fixing are satisfied in the Top-K generated pairs. We choose $K = 1, 5, 10$ for measuring the triggering and fixing rates.

Parameter and Hardware Settings. We split 80% of IRs with code commits for auto-prompting, and the rest 20% and the IRs w/o commits for evaluation. We fine-tune all the baselines (except for ChatGPT) with *batch_size* = 8. All experiments are run on a PC with Windows 11 OS, NVIDIA GeForce RTX 2060.

5 Result

5.1 Performances on Insecure Code Generation

We introduce the PatUntrack to improve the T5, GPT-3, and ChatGPT's performances, and the model names are LLMs+PatUntrack. In the evaluation dataset with code labels, we analyze the matching rate between generated insecure code and the ground-truth labels of insecure code. In the evaluation dataset without code labels, we ❶ first utilize the open-sourced security testing tools, such as Zed [14] and Wapiti [13], etc., to test whether the generated insecure code example will trigger the corresponding vulnerabilities, and ❷ manually test the insecure code if the automatic detectors cannot trigger the vulnerabilities.

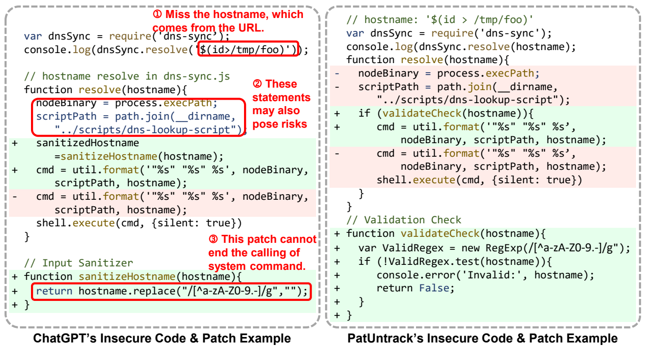
Table 4: The performances of baseline comparison on generating insecure code examples from vulnerable IRs (%).

Exp	Category	Model	Version	MatchLine	MatchTrig	AccType
w/ Code Labels	Non-LLM	CodeBert	-	16.6	35.9	-
	CodeGen	Codeium	1.6.10	45.2	60.7	-
	T5	CodeT5	t5-base	18.2	38.7	66.3
		+PatUntrack	t5-base	57.3 (↑39.1)	68.2 (↑29.5)	78.6 (↑12.3)
	GPT-3	Codex	davinci-003	27.5	50.2	62.9
		+PatUntrack	davinci-003	66.2 (↑38.7)	79.5 (↑29.3)	80.7 (↑17.8)
	GPT-3.5	ChatGPT	turbo-3.5	40.9	62.1	80.5
		+PatUntrack	turbo-3.5	**74.3** (↑33.4)	**81.0** (↑18.9)	**83.9** (↑3.4)
Exp	**Category**	**Model**	**Version**	**Trig@1**	**Trig@5**	**Trig@10**
w/o Code Labels	Non-LLM	CodeBert	-	39.2	50.4	61.3
	CodeGen	Codeium	1.6.10	20.5	44.6	47.9
	T5	CodeT5	t5-base	37.2	55.6	63.2
		+PatUntrack	t5-base	64.2 (↑27.0)	68.6 (↑13.0)	73.2 (↑10.0)
	GPT-3	Codex	davinci-003	58.5	63.4	65.2
		+PatUntrack	davinci-003	72.6 (↑14.1)	74.2 (↑10.8)	77.5 (↑12.3)
	GPT-3.5	ChatGPT	turbo-3.5	67.0	68.5	71.3
		+PatUntrack	turbo-3.5	**73.5** (↑6.5)	**76.9** (↑8.4)	**80.6** (↑9.3)

Comparison Results. Table 4 illustrates the results of PatUntrack on generating insecure code examples. Comparing the PatUntrack with all the code generation and LLM baselines, we can see that, the ChatGPT+PatUntrack can obtain the highest performances with 74.3% (MatchLine), 81.0% (MatchTrig), and 80.6% (Trig@10), improving LLM baselines with +37.1% (MatchLine), +25.9% (MatchTrig), and +10.5% (Trig@10) on average. Moreover, PatUntrack also improves LLM baseline's accuracy in predicting the vulnerability types with +11.2% on average.

Case Study. We conduct the case study on the vulnerable IR in Figure 2 to qualitatively evaluate the PatUntrack. Figure 6 shows the generated insecure code example of ChatGPT and +PatUntrack. Referring to the ground-truth insecure code in the repository, we can see that PatUntrack can accurately indicate that the `hostname` comes from the external URL and find all the statements that may contain CWE-78 vulnerabilities. ChatGPT has made errors in these two points and generated inaccurate code examples. These results illustrate that our approach is more accurate than baselines on generating insecure code examples.

Advantages of PatUntrack. We believe that the benefits of PatUntrack come from three aspects. ❶ First, PatUntrack can obtain the description of how to trigger a vulnerability, and the VTP extractor and VTP completer improve the completeness of



```
  var dnsSync = require('dns-sync');
  console.log(dnsSync.resolve('$(id>/tmp/foo)'));

  // hostname resolve in dns-sync.js
  function resolve(hostname){
    nodeBinary = process.execPath;
    scriptPath = path.join(__dirname,
      "../scripts/dns-lookup-script");
+   sanitizedHostname
      =sanitizeHostname(hostname);
+   cmd = util.format('"%s" "%s" %s', nodeBinary,
      scriptPath, hostname);
-   cmd = util.format('"%s" "%s" %s', nodeBinary,
      scriptPath, hostname);
    shell.execute(cmd, {silent: true})
  }

  // Input Sanitizer
+ function sanitizeHostname(hostname){
+   return hostname.replace("/[^a-zA-Z0-9.-]/g","");
+ }
```

① Miss the hostname, which comes from the URL.
② These statements may also pose risks
③ This patch cannot end the calling of system command.
ChatGPT's Insecure Code & Patch Example

```
  // hostname: '$(id > /tmp/foo)'
  var dnsSync = require('dns-sync');
  console.log(dnsSync.resolve(hostname);
  function resolve(hostname){
-   nodeBinary = process.execPath;
-   scriptPath = path.join(__dirname,
      "../scripts/dns-lookup-script");
+   if (validateCheck(hostname)){
+     cmd = util.format('"%s" "%s" %s',
        nodeBinary, scriptPath, hostname);
-     cmd = util.format('"%s" "%s" %s',
        nodeBinary, scriptPath, hostname);
      shell.execute(cmd, {silent: true})
    }
  }
  // Validation Check
+ function validateCheck(hostname){
+   var ValidRegex = new RegExp(/[^a-zA-Z0-9.-]/g");
+   if (!ValidRegex.test(hostname)){
+     console.error('Invalid:', hostname);
+     return False;
+   }
+ }
```

Figure 6: The case study of ChatGPT+PatUntrack and ChatGPT on generating insecure code & patch example.

generated VTP, which can facilitate the LLMs to generate code that reflects the vulnerabilities. ❷ Second, the hallucination correction can reduce the VTP descriptions that do not reflect real-world vulnerabilities. ❸ Third, the CWE and error type in the VTP description help LLM to accurately analyze the type of vulnerability.

> ***Answering RQ1****: ChatGPT+PatUntrack achieves the highest performances with 74.3% (MatchLine), 81.0% (MatchTrig), and 80.6% (Trig@10). Moreover, PatUntrack improves LLM baselines with +37.1% (MatchLine) and +25.9% (MatchTrig), and +10.5% (Trig@10) on average.*

5.2 Performances on Patch Example Generation

The experiment settings on the evaluation dataset with/without code labels are the same as Section 5.1. To keep Non-LLM APR baselines consistent with PatUntrack, they generate the patches based on the generated insecure code example of PatUntrack.

Table 5: The performances of baseline comparison on generating patch examples from vulnerable IRs (%).

Exp	Category	Model	Version	MatchLine	MatchFix	AccType
w/ Code Labels	Non-LLM	CodeBert	-	41.9	56.5	-
	APR	InCoder	InCoder-6.8B	46.5	67.3	-
	T5	CodeT5	t5-base	40.7	59.7	68.9
		+PatUntrack	t5-base	51.2 (↑10.5)	74.0 (↑14.3)	77.2 (↑8.3)
	GPT-3	Codex	davinci-003	53.0	62.2	76.2
		+PatUntrack	davinci-003	59.4 (↑6.4)	80.5 (↑18.3)	84.9 (↑8.7)
	GPT-3.5	ChatGPT	turbo-3.5	56.2	63.2	83.4
		+PatUntrack	turbo-3.5	**65.5** (↑9.3)	**83.7** (↑20.5)	**87.2** (↑3.8)
Exp	**Category**	**Model**	**Version**	**Fix@1**	**Fix@5**	**Fix@10**
w/o Code Labels	Non-LLM	CodeBert	-	38.4	51.5	54.2
	APR	InCoder	InCoder-6.8B	56.6	60.7	62.1
	T5	CodeT5	t5-base	40.5	55.2	56.4
		+PatUntrack	t5-base	51.5 (↑11.0)	60.0 (↑4.8)	67.1 (↑10.7)
	GPT-3	Codex	davinci-003	46.5	57.3	59.1
		+PatUntrack	davinci-003	66.2 (↑19.7)	72.3 (↑15.0)	75.9 (↑16.8)
	GPT-3.5	ChatGPT	turbo-3.5	50.3	61.7	62.3
		+PatUntrack	turbo-3.5	**69.7** (↑19.4)	**73.2** (↑11.5)	**78.5** (↑16.2)

Comparison Results. Table 5 illustrates the comparison results of PatUntrack on generating patch examples from IR textual description. Comparing the PatUntrack with all the code generation and LLM baselines, we can see that, the ChatGPT+PatUntrack can obtain the highest performances with 65.5% (MatchLine), 83.7% (MatchFix), and 78.5% (Fix@10), improving LLM baselines with +8.7% (MatchLine), +17.7% (MatchFix), and +14.6% (Fix@10) on average. Moreover, PatUntrack also improves LLM's accuracy in predicting the patch types with +6.9% on average.

Case Study. Figure 6 also shows the results of the case study on patch example generation. We can see that the patch example generated by PatUntrack can successfully utilize the input validation to fix the vulnerability. ChatGPT utilizes the `sanitizer` to replace the error strings with `null` but does not terminate the system command. If the regular expression cannot match the whole input string, the attack may still succeed. Therefore, the PatUntrack's patch example is more appropriate to fix the vulnerabilities.

Advantages of PatUntrack. In addition to the advantages in Section 5.1, the benefits of PatUntrack also come from the patch type prediction and the joint prediction. These methods can reduce the biases in generated patch examples, thus improving the accuracy and fixing rate of patch generation.

> ***Answering RQ2**: ChatGPT+PatUntrack achieves the highest performances with 65.5% (MatchLine), 83.7% (MatchFix), and 78.5% (Fix@10). Moreover, PatUntrack improves LLM baselines with +8.7% (MatchLine), +17.7% (MatchFix), and +14.6% (Fix@10) on average.*

5.3 Effect of IR's Information

In Figure 4, we introduce the focus list to guide the generation of patch examples when the IR lacks detailed information, and we also complete the missing nodes and edges in the VTP description. In this experiment, we measure how the PatUntrack depends on IR's amount of information. Since there is no standard way to measure the richness of IR's information, we report the number of iterations (abbreviated as Iter.) in Section 3.1, where each iteration is a step with **VTP Extraction⇆VTP Completing**. This iteration indicates the difficulty of generating a completed VTP description. Intuitively, an IR with less information would require more iterations to generate a complete VTP description.

Based on the iterations, we split the evaluation IRs (**3,473 #Evaluation** in Table 3) into three intervals, i.e., **1≤Iter.<4** (1,409/3,473 IRs), **4≤Iter.<8** (1,161/3,473 IRs), and **Iter.≥8** (903/3,473 IRs). We can see that 26% of them are minimally descriptive IRs (e.g., a single sentence to describe the vulnerability but lacks details), where the VTP generator takes ≥8 iterations to complete the missing nodes and edges in the generated VTP descriptions.

We compare the performances of LLMs and LLM+PatUntrack on the metrics of insecure code example generation (i.e., MatchTrig and Trig@10), and patch example generation (i.e., MatchFix and Fix@10) within these three iteration intervals. We can see that the performances of all techniques are decreased when the information in the IR is less detailed. However, PatUntrack can accurately generate the insecure code & patch examples, outperforming these original LLMs, with over +14.1% (Trig@10) and +27.3% (Fix@10) in Iter.≥8. Moreover, PatUntrack has fewer fluctuations than original LLMs among different iteration intervals. From 1≤Iter.<4 to Iter.≥8, the performances of PatUntrack decrease by less than ±5.0% (Trig@10) and ±6.0% (Fix@10). These results illustrate the ability of PatUntrack to handle IRs that lack detailed information.

Table 6: The performances on generating insecure code & patch examples with different numbers of iterations(%).

Iterations	Category	Models	Insecure Code Example		Patch Example	
			MatchTrig	Trig@10	MatchFix	Fix@10
1≤Iter.<4	**T5**	CodeT5	39.5	66.2	64.1	59.5
		+PatUntrack	72.2 (↑32.7)	76.0 (↑9.8)	76.0 (↑11.9)	70.6 (↑11.1)
	GPT-3	Codex	56.5	66.4	64.9	63.2
		+PatUntrack	**83.1** (↑26.6)	78.6 (↑12.2)	82.4 (↑17.5)	77.2 (↑14.0)
	GPT-3.5	ChatGPT	70.5	82.0	66.4	67.4
		+PatUntrack	82.5 (↑12.0)	**84.2** (↑2.2)	**85.0** (↑18.6)	**81.3** (↑13.9)
4≤Iter.<8	**T5**	CodeT5	38.6	63.5	61.2	58.4
		+PatUntrack	68.1 (↑29.5)	72.2 (↑8.7)	73.0 (↑11.8)	69.2 (↑10.8)
	GPT-3	Codex	50.7	62.1	62.5	58.5
		+PatUntrack	**82.4** (↑31.7)	78.1 (↑16.0)	80.7 (↑18.2)	76.4 (↑17.9)
	GPT-3.5	ChatGPT	61.5	78.4	65.5	61.5
		+PatUntrack	81.9 (↑20.4)	**81.9** (↑3.5)	**85.6** (↑20.1)	**79.2** (↑17.7)
Iter.≥8	**T5**	CodeT5	11.0	31.9	23.5	28.7
		+PatUntrack	67.4 (↑56.4)	71.5 (↑39.6)	73.2 (↑49.7)	65.4 (↑36.7)
	GPT-3	Codex	25.6	36.7	39.0	30.0
		+PatUntrack	76.2 (↑50.6)	76.9 (↑40.2)	76.4 (↑37.4)	73.6 (↑43.6)
	GPT-3.5	ChatGPT	43.7	65.4	54.2	49.5
		+PatUntrack	**80.1** (↑36.4)	**79.5** (↑14.1)	**80.2** (↑26.0)	**76.8** (↑27.3)

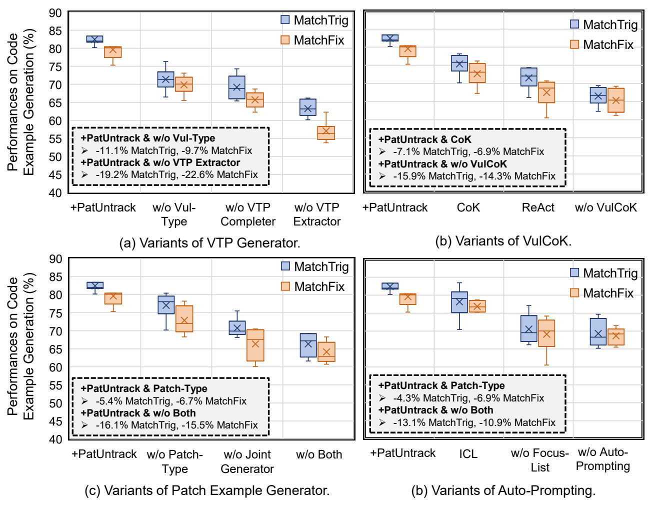


Figure 7: The results of the ablation study (PatUntrack and variants utilize the ChatGPT as the basic model).

> ***Answering RQ3**: PatUntrack can handle the IRs when they lack detailed information. It outperforms LLM baselines with over +14.1% (Trig@10) and +27.3% (Fix@10) when Iter.≥8. It also has the fewer fluctuations among iteration intervals with less than ±5% (Trig@10) and ±6% (Fix@10).*

5.4 Ablation Study

In the ablation study, we conduct experiments on four types of variants, i.e., VTP generator, VulCoK, patch example generator, and auto-prompting. We compare the performances of PatUntrack and variants on **MatchTrig** and **MatchFix**:

- **VTP Generator:** The variants are removing the Vul_Type, VTP completer, and the whole VTP extractor.
- **VulCoK:** The variants are replacing VulCoK with CoK/ReAct, and removing the whole VulCoK.
- **Patch Example Generator:** The variants are removing the Vul_Type, Joint Generation, and both variants.
- **Auto-Prompting:** The variants are replacing VulCoK with In-Context Learning (ICL) [53], which is a representative method that optimizes ChatGPT with relevant samples, and removing the Focus-List or the whole Auto-Prompting.

Comparison on VTP Generator. Figure 7 (a) shows comparison results of the VTP generator. We can see that removing vulnerable types leads to a moderate decrease of -11.1% (MatchTrig) and -9.7% (MatchFix); removing the whole VTP extractor leads to the largest decrease with -19.2% (MatchTrig) and -22.6% (MatchFix).

Comparison on VulCoK. Figure 7 (b) shows comparison results of VulCoK. We can see that replacing it with normal CoK leads to a moderate decrease of -7.1% (MatchTrig) and -6.9% (MatchFix); removing the whole VulCoK leads to the largest decrease with -15.9% (MatchTrig) and -14.3% (MatchFix).

Comparison on Patch Example Generator. Figure 7 (c) shows comparison results of the insecure code & patch generator. We can see that removing patch types leads to a moderate decrease with -5.4% (MatchTrig) and -6.7% (MatchFix); removing both patch type and joint generation leads to the largest decrease with -16.1% (MatchTrig) and -15.5% (MatchFix).

Comparison on Auto-Prompting Figure 7 (d) shows the comparison results of auto-prompting. We can see that replacing it with ICL leads to a moderate decrease of -4.3% (MatchTrig) and -6.9% (MatchFix); removing the auto-prompting leads to the largest decrease with -13.1% (MatchTrig) and -10.9% (MatchFix).

***Answering RQ4**: PatUntrack outperforms all the variants in ablation study. Removing the VTP extractor, VulCoK, joint patch generator, and auto-prompting leads to the largest decrease on MatchTrig and MatchFix; removing vulnerability/patch types and replacing components with CoK/ICL leads to a moderate decrease.*

6 Human Evaluation

To analyze the performances of PatUntrack on generating patch examples for the vulnerable IRs without insecure code, we track the vulnerable IRs from GitHub that are **not included in the collected dataset** in Section 4.1. We track the IRs with the issue-tracking system [64] Then, we observe that 76 CVE-disclosed vulnerable IRs have not released the code commits, which may pose security risks to the public. To analyze the contribution of PatUntrack, we utilize the ChatGPT and ChatGPT+PatUntrack to generate the patch examples and ask the authors the following questions:

- ***Q1:*** *Does the type of patch example we provide match the vulnerabilities you have encountered? Please reply: {Yes/No}.*
- ***Q2:*** *Will you accept the patch example to help you fix the vulnerabilities in your projects? Please reply: {Accepted/Unaccepted}.*
- ***Q3:*** *Please use several sentences to describe the reason why the patch example can/cannot be accepted to fix the vulnerabilities.*

We contact the IR authors with *Emails* or directly submit the comments on the IR pages. Then, we analyze the proportion of acceptance. Table 7 shows the number and ratio of accepted insecure code & patch examples. We received 37 of 76 responses (48.7%) from the IR authors, and 27 pairs (35.5%) generated by ChatGPT+PatUntrack can help authors fix the vulnerabilities, which is +15.8% higher than ChatGPT. Most insecure code & patch example pairs (20) belong to CWE-79, and 9 pairs (45.0%) are accepted.

Moreover, we ask the authors to manually inspect the generated results and rate their "*accurateness*". This criterion measures the accuracy of the generated insecure code & patch examples to the real commits in their projects. We ask the authors to rate 1-5 under the above criteria. For each accepted/unaccepted patch example, a score of 5 means that the generated code almost matches the real code in the repository, and a score of 1 means the generated code is completely different from the real code. A score of 3 is borderline, which means the generated code only needs a few modifications to become satisfactory code and reflect the vulnerabilities.

Table 7: The number (#) and the ratio of accepted generated insecure code & patches for ChatGPT and PatUntrack.

CWE-Types	#Total-Pairs	#Response-Pairs	#Acc-PatUntrack	#Acc-ChatGPT	#Acc-Both
CWE-79	20	12 (60.0%)	9 (45.0%)	4 (20.0%)	2 (10.0%)
CWE-787	17	8 (47.1%)	4 (23.5%)	4 (23.5%)	3 (17.6%)
CWE-78	10	7 (70.0%)	4 (40.0%)	2 (20.0%)	2 (20.0%)
CWE-352	8	3 (37.5%)	3 (37.5%)	1 (12.5%)	1 (12.5%)
CWE-287	8	3 (37.5%)	2 (25.0%)	2 (25.0%)	2 (25.0%)
CWE-121	7	2 (28.6%)	2 (28.6%)	1 (14.3%)	1 (14.3%)
CWE-119	6	2 (33.3%)	3 (50.0%)	1 (16.7%)	1 (16.7%)
Total	*76*	*37 (48.7%)*	*27 (35.5%)*	*15 (19.7%)*	*12 (15.8%)*

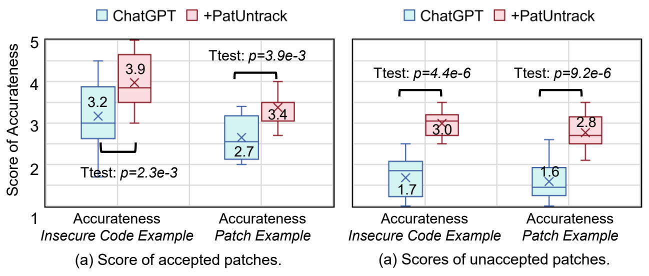


Figure 8: The score distribution of human evaluation (PatUntrack utilizes the ChatGPT as the basic model).

Figure 8 shows the scores of accepted and unaccepted pairs for both ChatGPT and ChatGPT+PatUntrack. For the accepted pairs, the scores of +PatUntrack are significantly higher than ChatGPT[1], and the improvement of average scores are both +0.7. For the unaccepted pairs, +PatUntrack also significantly outperforms the ChatGPT; the improvement scores are +1.3 and +1.2 on average. The average scores of +PatUntrack are nearly 3.0, which means the unaccepted code only needs a few corrections to be accepted. These results illustrate that PatUntrack can be practically utilized to help authors fix their vulnerabilities.

7 Discussion

7.1 Effect of Joint Code Generation

In Section 3.3, we utilize the joint code generation to generate the insecure code & patch examples accurately. To analyze the effect of insecure code examples on the patch example generation during the auto-prompting, we compare the performances of ChatGPT and ChatGPT+PatUntrack on the change of **MatchTrig→MatchFix** and **Trig@10→Fix@10** on our evaluation dataset.

Figure 9 shows the generation results of insecure code & patch examples in the auto-prompting. We can see that, with the improvement of MatchTrig and Trig@10, both ChatGPT and PatUntrack's generation results on patch examples will improve, and the differences between these models will also increase. When MatchTrig and Trig@10 reach 60%~65%, PatUntrack's MatchFix and Fix@10 reach the maximum value, which means patch examples are appropriate based on the insecure code examples.

7.2 Unsuccessful Cases

Although PatUntrack can generate appropriate insecure code & patch examples, there are still 10%~15% cases, where PatUntrack

[1]$p < 0.05$ in the T-test shows the significant differences between two sets of data.

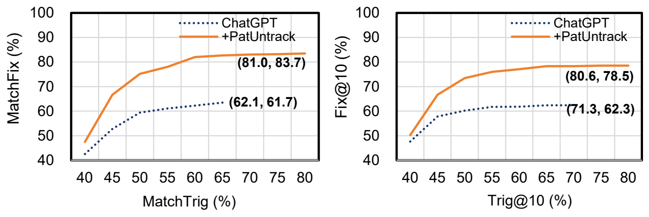


Figure 9: Insecure code's effect to patch example generation.

failed to generate correct patches. We manually inspected these unsuccessful cases and found that most of these cases come from incompletely generated VTPs. Some IR authors omit too many details in the IR descriptions, so PatUntrack cannot complete the VTP operations and transitions solely based on the IR descriptions. For example, the *bedita/bedita/issues/755* [2] (CVE-2015-9260) does not provide any description of the vulnerability except for links to external reports that are currently not analyzed by PatUntrack, thus PatUntrack cannot generate insecure code & patch examples based on IR description.

7.3 Threats to Validity

Internal Threats. The internal threat mainly comes from the approach. First, we only analyze the textual description in IR to build the VTP description, and we plan to utilize other sources, such as textual description in the third-party links to improve the PatUntrack. Second, the time costs of loops in the VTP generator and VulCoK are heavy, which may affect its practical usage. To alleviate it, we set an upper limit $\theta = 10$ for auto-prompting, and the current step will exit if the loops exceed this limitation. Third, we utilize the LLM to correct hallucinatory types and descriptions, which may have biases in the VulCoK. In the future, we plan to analyze the success rate of hallucination correction to mitigate this threat.
External Threat. The external threat may come from the dataset we use. We only refer to the CVE-disclosed IRs to analyze whether these IRs contain the vulnerabilities. However, some vulnerable IRs may not be incorporated by these two datasets, and some vulnerabilities may be disclosed by other security databases, such as CAPEC [22]. Another threat comes from the silent patches, since the authors may quietly submit patches for the vulnerabilities but not report them to the public. To alleviate it, we manually inspected 100 vulnerable IRs from the public and found that only 7 of them have these threats, so the impact of external threats is small.
Constructive Threat. The constructive threat mainly comes from the metrics. All the chosen metrics, i.e., MatchLine, MatchTrig/-MatchFix, and Trig&K/Fix@K, may have biases for evaluating the generated results. We review and discuss the settings of metrics with team members, thus alleviating this constructive threat.

8 Related Works

Vulnerability Detection and Analysis from OSS Projects. Recently, researchers have proposed various approaches to detect and analyze the vulnerabilities in OSS projects. Automatic vulnerability detection aims to determine whether there are malicious code in the projects [30, 36, 38, 46, 48, 51, 96]. The researchers first proposed the statistic, dynamic, and hybrid techniques to detect the vulnerabilities with rules [24, 35, 45]. With the development of machine learning (ML) and Deep Learning (DL) approaches, researchers utilized these novel models to automatically build code features and improve the efficiency of vulnerability detection tools [44, 44, 47, 71, 74, 79, 88]. In addition, researchers also analyze the vulnerabilities from various project artifacts (e.g., IRs, bug reports, etc.). Some researchers utilized text-mining methods to explore the security bug reports to identify the vulnerabilities [29, 82–84], while other works analyze the negative impact of the vulnerabilities from the IRs [62, 64, 66, 75]. The other researchers focus on the crowd-based security discussions, e.g., security posts in Stack Overflow, and discussion groups in Gitter/Slacks, to analyze the topics, attacks, and the corresponding mitigations [40, 52, 67, 89, 92]. Different from these previous works, we build the gaps between the IR textual description and source code by generating insecure code & patch examples that cannot track the insecure code, thus helping developers fix the vulnerabilities.
Patch Generation for Vulnerable OSS Projects. APR methods are the typical methods for generating patches for fixing normal bugs or vulnerabilities. The template-based APR methods leverage different bug-fixing templates, which are designed by human experts, to fix the specific types of bugs in the source code [31, 32, 49]. Recent researchers have proposed learning-based APR tools, which typically model program repair as a Neural Machine Translation (NMT) problem [37, 50, 90, 95]. With the development of LLM, the researchers also analyze how to combine the LLMs to the APR tools to improve their patching ability [85], and reduce the time and financial cost of LLM [86]. These works rely on the source code to fix the vulnerabilities, which cannot be applied on IRs without tracked insecure code. On the contrary, our approach can generate patch examples based on IR textual description, which can timely help developers fix the vulnerabilities after the IR creation.

9 Conclusion

In this paper, we introduced PatUntrack to generate patch examples from IRs without tracked insecure code. It auto-prompts LLMs to make them applicable for analyzing the vulnerabilities described in IRs and generating appropriate patch examples. Specifically, it first generates the completed VTP description from vulnerable IRs. Then, it utilizes the VulCoK to correct the hallucinatory VTP description. Finally, it generates Top-K pairs of *Insecure Code and Patch Example* based on the corrected VTP description. Experiments conducted on 5,465 vulnerable IRs show that PatUntrack can achieve the highest performance and improve the traditional LLM baselines by +17.7% (MatchFix) and +14.6% (Fix@10) on average in patch example generation. Furthermore, PatUntrack has been applied to generating patch examples for 76 newly disclosed vulnerable IRs, and 27 out of 37 replies from the authors of these IRs confirmed the usefulness of the patch examples generated by PatUntrack, indicating that they can benefit from these examples for patching the vulnerabilities.

In the future, we plan to enhance the PatUntrack by introducing other third-party resources to generate the VTP descriptions, as well as analyzing whether VTP can improve the performance of traditional APR tools in patch generation.

Acknowledgments

We sincerely appreciate the reviewers for their constructive and insightful suggestions. This work is supported by the National Natural Science Foundation of China Grant No. 62332001, 62272445, 62232016, 62072442, the Youth Innovation Promotion Association Chinese Academy of Sciences, Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202304, Major Program of ISCAS Grant No. ISCAS-ZD-202302, partially supported by the University of Queensland NSRSG Grant No. NS-2201 and Oracle Labs.

References

[1] 2014. This package contains a serious security hole. https://github.com/skoranga/node-dns-sync/issues/1.

[2] 2015. Bedita CMS 3.6.0 – Publication Module Bug Report. https://github.com/bedita/bedita/issues/755.

[3] 2018. ISO/IEC 29147:2018: Security techniques - Vulnerability disclosure. https://www.iso.org/standard/72311.html..

[4] 2018. Possible XSS in safe_mode using incomplete tags. https://github.com/trentm/python-markdown2/issues/285.

[5] 2023. Bugzilla. https://www.bugzilla.org/.

[6] 2023. Common vulnerabilities and exposures. https://cve.mitre.org/.

[7] 2023. Common weakness enumeration. https://cwe.mitre.org/.

[8] 2023. GHArchive. https://www.gharchive.org/.

[9] PatUntrack. 2024. PatUntrack. https://www.doi.org/10.6084/m9.figshare.26643037.

[10] 2024. CERT Guide to Coordinated Vulnerability Disclosure. https://vuls.cert.org/confluence/display/CVD.

[11] 2024. Codeium - Free AI Code Completion & Chat. https://codeium.com/.

[12] 2024. google-t5/t5-base. https://huggingface.co/google-t5/t5-base.

[13] 2024. Wapiti - A simple and fast discriminative sequence labelling toolkit. https://wapiti.limsi.fr/.

[14] 2024. Zed Attack Proxy. https://www.zaproxy.org/.

[15] Zeeshan Afzal, Johan Garcia, Stefan Lindskog, and Anna Brunström. 2018. Slice Distance: An Insert-Only Levenshtein Distance with a Focus on Security Applications. In *9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018*. IEEE, 1–5.

[16] Atlassian. 2023. Jira | Issue & Project Tracking Software. https://www.atlassian.com/software/jira.

[17] Leyla Bilge and Tudor Dumitras. 2012. Before we knew it: an empirical study of zero-day attacks in the real world. In *the ACM Conference on Computer and Communications Security, CCS'12*. ACM, 833–844.

[18] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*. ACM, 654–668.

[19] Xiao Cheng, Xu Nie, Ningke Li, Haoyu Wang, Zheng Zheng, and Yulei Sui. 2024. How About Bug-Triggering Paths? - Understanding and Characterizing Learning-Based Vulnerability Detectors. *IEEE Trans. Dependable Secur. Comput.* 21, 2 (2024), 542–558.

[20] Wen-Hao Chiang, Peixuan Li, Qiang Zhou, Subarno Banerjee, Martin Schäf, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. Inference for Ever-Changing Policy of Taint Analysis. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024*. ACM, 452–462.

[21] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 87:1–87:13.

[22] T. M. Corporation. 2011. Common Attack Pattern Enumeration and Classification (CAPEC). *http://capec. mitre.org/* (2011).

[23] Clément Elbaz, Louis Rilling, and Christine Morin. 2020. Automated Keyword Extraction from "One-day" Vulnerabilities at Disclosure. In *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–9.

[24] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*. ACM, 57–72.

[25] Explosion. 2022. Spacy. https://www.spacy.io/.

[26] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. 2024. LLM Agents can Autonomously Exploit One-day Vulnerabilities. *CoRR* abs/2404.08144 (2024).

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547.

[28] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*.

[29] Michael Gegick, Pete Rotella, and Tao Xie. 2010. Identifying security bug reports via text mining: An industrial case study. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE)*. IEEE Computer Society, 11–20.

[30] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–36.

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

[32] Ali Ghanbari and Lingming Zhang. 2019. PraPR: Practical Program Repair via Bytecode Mutation. In *34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1118–1121.

[33] Allen D Householder, Garret Wassermann, Art Manion, and Chris King. 2017. The cert guide to coordinated vulnerability disclosure. *Software Engineering Institute, Pittsburgh, PA* (2017).

[34] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *CoRR* abs/2311.05232 (2023).

[35] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *IEEE Symposium on Security and Privacy, SP 2012*. IEEE Computer Society, 48–62.

[36] Tiantian Ji, Yue Wu, Chang Wang, Xi Zhang, and Zhongru Wang. 2018. The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques. In *2018 IEEE third international conference on data science in cyberspace (DSC)*. IEEE, 53–60.

[37] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*. IEEE, 1161–1173.

[38] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. 2016. Survey on software vulnerability analysis method based on machine learning. In *2016 IEEE first international conference on data science in cyberspace (DSC)*. IEEE, 642–647.

[39] Triet Huynh Minh Le, Roland Croft, David Hin, and Muhammad Ali Babar. 2020. Demystifying the Mysteries of Security Vulnerability Discussions on Developer Q&A Sites. *CoRR* abs/2008.04176 (2020).

[40] Triet Huynh Minh Le, Roland Croft, David Hin, and Muhammad Ali Babar. 2021. A Large-Scale Study of Security Vulnerability Support on Developer Q&A Websites. In *Evaluation and assessment in software engineering*. 109–118.

[41] Mingyang Li, Lin Shi, Ye Yang, and Qing Wang. 2020. A Deep Multitask Learning Approach for Requirements Discovery and Annotation from Open Forum. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 336–348.

[42] Xingxuan Li, Ruochen Zhao, Yew Ken Chia, Bosheng Ding, Lidong Bing, Shafiq R. Joty, and Soujanya Poria. 2023. Chain of Knowledge: A Framework for Grounding Large Language Models with Structured Knowledge Bases. *CoRR* abs/2305.13269 (2023).

[43] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258.

[44] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS*.

[45] Hongliang Liang, Lei Wang, Dongyang Wu, and Jiuyun Xu. 2016. MLSA: A static bugs analysis tool based on LLVM IR. In *17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2016, Shanghai, China, May 30 - June 1, 2016*. IEEE Computer Society, 407–412.

[46] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* 108, 10 (2020), 1825–1848.

[47] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2539–2541.

[48] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*. IEEE, 152–156.

[49] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42.

[50] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114.

[51] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[52] Benjamin S Meyers, Nuthan Munaiah, Andrew Meneely, and Emily Prud'hommeaux. 2019. Pragmatic Characteristics of Security Conversations:

An Exploratory Linguistic Analysis. In *CHASE*. IEEE, 79–82.

[53] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022*. Association for Computational Linguistics, 11048–11064.

[54] MITRE. 2024. CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). https://cwe.mitre.org/data/definitions/78.html.

[55] Van Nguyen, Trung Le, Olivier Y. de Vel, Paul Montague, John Grundy, and Dinh Phung. 2021. Information-theoretic Source Code Vulnerability Highlighting. In *International Joint Conference on Neural Networks, IJCNN 2021*. IEEE, 1–8.

[56] Devon H. O'Dell. 2017. The Debugging Mindset. *ACM Queue* 15, 1 (2017), 50.

[57] Marwan Omar and Stavros Shiaeles. 2023. VulDetect: A novel technique for detecting software vulnerabilities using Language Models. In *IEEE International Conference on Cyber Security and Resilience, CSR 2023, Venice, Italy, July 31 - Aug. 2, 2023*. IEEE, 105–110.

[58] OpenAI. 2023. Chatgpt: A language model for conversational AI. https://www.openai.com/research/chatgpt/.

[59] OpenAI. 2024. GPT35Models. https://platform.openai.com/docs/models/gpt-3-5.

[60] OpenAI. 2024. GPT3Models. https://platform.openai.com/docs/models/gpt-3.

[61] OWASP. 2023. Open web application security project. https://www.owasp.org/index.php/MainPage.

[62] Tosin Daniel Oyetoyan and Patrick Morrison. 2021. An improved text classification modelling approach to identify security messages in heterogeneous projects. *Softw. Qual. J.* 29, 2 (2021), 509–553.

[63] Liuxuan Pan and Allan Tomlinson. 2016. A Systematic Review of Information Security Risk Assessment. *International Journal of Safety and Security Engineering* 6 (06 2016), 270–281.

[64] Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Côgo, Xin Xia, Lingfeng Bao, Xing Hu, Shanping Li, and Ahmed E. Hassan. 2022. Automated unearthing of dangerous issue reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 834–846.

[65] Jorge E. Pérez, Jessica Díaz, Javier García Martin, and Bernardo Tabuenca. 2020. Systematic Literature Reviews in Software Engineering - Enhancement of the Study Selection Process Using Cohen's Kappa Statistic. *J. Syst. Softw.* 168 (2020), 110657.

[66] Fayola Peters, Thein Than Tun, Yijun Yu, and Bashar Nuseibeh. 2019. Text Filtering and Ranking for Security Bug Report Prediction. *IEEE Trans. Software Eng.* 45, 6 (2019), 615–631.

[67] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. 2014. Security and Emotion: Sentiment Analysis of Security Discussions on Github. In *Proceedings of the 11th working conference on mining software repositories*. 348–351.

[68] Hang Ruan, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2019. DeepLink: Recovering issue-commit links based on deep learning. *J. Syst. Softw.* 158 (2019).

[69] Gordon Rugg and Peter McGeorge. 1997. The Sorting Techniques: a Tutorial Paper on Card Sorts, Picture Sorts and Item Sorts. *Expert Systems* 14, 2 (1997), 80–93.

[70] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*. IEEE, 757–762.

[71] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. *IEEE Trans. Software Eng.* 40, 10 (2014), 993–1006.

[72] Lin Shi, Ziyou Jiang, Ye Yang, Xiao Chen, Yumin Zhang, Fangwen Mu, Hanzhi Jiang, and Qing Wang. 2021. ISPY: Automatic Issue-Solution Pair Extraction from Community Live Chats. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 142–154.

[73] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 4222–4235.

[74] Yonghee Shin, Andrew Meneely, Laurie A. Williams, and Jason A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Trans. Software Eng.* 37, 6 (2011), 772–787.

[75] Rui Shu, Tianpei Xia, Jianfeng Chen, Laurie A. Williams, and Tim Menzies. 2021. How to Better Distinguish Security Bug Reports (Using Dual Hyperparameter Optimization). *Empir. Softw. Eng.* 26, 3 (2021), 53.

[76] Laurent Siklóssy, A. Rich, and Vesko Marinov. 1973. Breadth-First Search: Some Surprising Results. *Artif. Intell.* 4, 1 (1973), 1–27.

[77] Tencent. 2023. OCR | Tencent Cloud. https://www.tencentcloud.com/document/product/1045/49147.

[78] James Walden, Maureen Doyle, Grant A. Welch, and Michael Whelan. 2009. Security of open source web applications. In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*. IEEE Computer Society, 545–553.

[79] Jin Wang, Zishan Huang, Hengli Liu, Nianyi Yang, and Yinhao Xiao. 2023. DefectHunter: A Novel LLM-Driven Boosted-Conformer-based Code Vulnerability Detection Mechanism. *CoRR* abs/2309.15324 (2023).

[80] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. PatchDB: A Large-Scale Security Patch Dataset. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021*. IEEE, 149–160.

[81] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 1069–1088.

[82] Dumidu Wijayasekara, Milos Manic, and Miles McQueen. 2014. Vulnerability identification and classification via text mining bug databases. In *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society, Dallas, TX, USA, October 29 - November 1, 2014*. IEEE, 3612–3618.

[83] Dumidu Wijayasekara, Milos Manic, Jason L. Wright, and Miles McQueen. 2012. Mining Bug Databases for Unidentified Software Vulnerabilities. In *2012 5th International Conference on Human System Interactions, Perth, Australia, June 6-8, 2012*. IEEE, 89–96.

[84] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 821–833.

[85] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023*. IEEE, 1482–1494.

[86] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *CoRR* abs/2304.00385 (2023).

[87] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. 2023. PUMM: Preventing Use-After-Free Using Execution Unit Partitioning. In *32nd USENIX Security Symposium, USENIX Security 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 823–840.

[88] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: exposing missing checks in source code for vulnerability discovery. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 499–510.

[89] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. *Journal of Computer Science and Technology* 31, 5 (2016), 910–924.

[90] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022*. ACM, 1506–1518.

[91] Kang Min Yoo, Dongju Park, Jaewook Kang, Sang-Woo Lee, and Woo-Myoung Park. 2021. GPT3Mix: Leveraging Large-scale Language Models for Text Augmentation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, 2225–2239.

[92] Mansooreh Zahedi, Muhammad Ali Babar, and Christoph Treude. 2018. An Empirical Study of Security Issues Posted in Open Source Projects. In *HICSS*. ScholarSpace / AIS Electronic Library (AISeL), 1–10.

[93] Chenyuan Zhang, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. EALink: An Efficient and Accurate Pre-Trained Framework for Issue-Commit Link Recovery. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023*. IEEE, 217–229.

[94] Yunhui Zheng, Saurabh Pujar, Burn L. Lewis, Luca Buratti, Edward A. Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021*. IEEE, 111–120.

[95] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353.

[96] Yuhui Zhu, Guanjun Lin, Lipeng Song, and Jun Zhang. 2023. The application of neural network for software vulnerability detection: a review. *Neural Comput. Appl.* 35, 2 (2023), 1279–1301.