Generating API Parameter Security Rules with LLM for API Misuse Detection

Jinghua Liu[1,2], Yi Yang[1,2,⋆], Kai Chen[1,2,⋆], and Miaoqian Lin[1,2]

[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
{liujinghua, yangyi, chenkai, linmiaoqian}@iie.ac.cn

***Abstract*—When utilizing library APIs, developers should follow the API security rules to mitigate the risk of API misuse. API Parameter Security Rule (APSR) is a common type of security rule that specifies how API parameters should be safely used and places constraints on their values. Failure to comply with the APSRs can lead to severe security issues, including null pointer dereference and memory corruption. Manually analyzing numerous APIs and their parameters to construct APSRs is labor-intensive and needs to be automated. Existing studies generate APSRs from documentation and code, but the missing information and limited analysis heuristics result in missing APSRs. Due to the superior Large Language Model's (LLM) capability in code analysis and text generation without predefined heuristics, we attempt to utilize it to address the challenge encountered in API misuse detection. However, directly utilizing LLMs leads to incorrect APSRs which may lead to false bugs in detection, and overly general APSRs that could not generate applicable detection code resulting in many security bugs undiscovered.**

In this paper, we present a new framework, named GPTAid, for automatic APSRs generation by analyzing API source code with LLM and detecting API misuse caused by incorrect parameter use. To validate the correctness of the LLM-generated APSRs, we propose an execution feedback-checking approach based on the observation that security-critical API misuse is often caused by APSRs violations, and most of them result in runtime errors. Specifically, GPTAid first uses LLM to generate raw APSRs and the Right calling code, and then generates Violation code for each raw APSR by modifying the Right calling code using LLM. Subsequently, GPTAid performs dynamic execution on each piece of Violation code and further filters out the incorrect APSRs based on runtime errors. To further generate concrete APSRs, GPTAid employs a code differential analysis to refine the filtered ones. Particularly, as the programming language is more precise than natural language, GPTAid identifies the key operations within Violation code by differential analysis, and then generates the corresponding concrete APSR based on the aforementioned operations. These concrete APSRs could be precisely interpreted into applicable detection code, which proven to be effective in API misuse detection. Implementing on the dataset containing 200 randomly selected APIs from eight popular libraries, GPTAid achieves a precision of 92.3%. Moreover, it generates 6 times more APSRs than state-of-the-art detectors on a comparison dataset of previously reported bugs and APSRs. We further evaluated GPTAid on 47 applications, 210 unknown security bugs were found potentially resulting in severe security issues (e.g., system crashes), 150 of which have been confirmed by developers after our reports.

⋆ Corresponding Author

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230465
www.ndss-symposium.org

```
01 //APSR: Users should release the second
02 //parameter when no longer needed
03 if(sqlite3_open(..., &db->handle))
04 { ...
05   g_free(dbname);
06   g_free(db);
07   return NULL;
08 }...
```

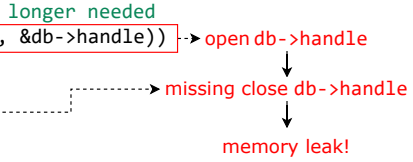

Fig. 1: Example for an API misuse in darktable

I. Introduction

Today, Application Programming Interfaces (APIs) play a vital role in software development, which enables developers to reuse functions from software libraries. API security rules should be strictly followed by software developers and could be classified as parameter rules (e.g., *"parameter must not be NULL"*), rules focusing on return value (e.g., *"return value must be checked against NULL"*) and the rules involving invocation condition (e.g., *"must be called before any other action takes place"*). Violating security rules can result in significant security issues, such as memory corruption, Denial-of-Service, and so on. API parameter security rules (APSRs) is one of the common types of security rules that have been extensively studied in the previous research [1], [2]. Specifically, APSRs specify the security rules on parameter values (e.g., *"parameters must not be negative"*) and the parameter-associated operations (e.g., *"must not be freed"*). According to the thorough analysis of 100 randomly selected known misuses from the existing work [3], [4], [5], [6], we found 71% of them resulted from APSR violations. For example, Figure 1 illustrates the misuse of the API `sqlite3_open` from the *darktable* application. One APSR of `sqlite3_open` says: *"release the second parameter when no longer needed"*. According to the figure, the second parameter `db->handle` is allocated in Line 3, however, the caller fails to release the allocated resource in case `sqlite3_open` fails within Line 5 to Line 7, which violates this APSR and leads to a memory

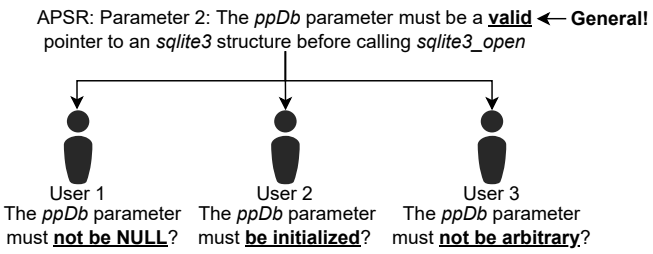

Fig. 2: Overly general APSR leads to incorrect interpretations

leak. With the extensive information from APSRs, automatic analyzers can detect API misuses and support secure software development, making automatic APSR generation essential.

Prior studies on APSRs generation and API misuse detection typically involve several limitations. Documentation-based approaches [6], [7], [1], [2], [8] generate APSRs based on the extracted knowledge, which may lead to security bugs undetected due to the absence of APSR description. The approaches based on the API calling code [9], [10], [11], [12], [8] generate APSRs by comparing usage patterns, which may also yield false negatives (Section V-D) resulting from the difficulties in identifying the correct patterns. Different from the former types of data, API source code is found to be a solid resource as it is the actual implementation of the API's functionality. Existing studies have shown its effectiveness [8], [4], [1] on API misuse detection. However, they are limited to specific bug types, due to the high reliance on predefined rules which are designed specifically for certain code statements. Unfortunately, the analysis of various statements within the API source code is time-consuming and needs complex data/-control flow analysis and natural language rules construction for each piece of analysis. With the superior capability of Large Language Models (LLMs) on code analysis and text generation, enabling the analysis of both programming and natural languages at the same time, we propose to exploit LLMs for APSRs generation.

Challenges in using LLMs. However, directly using LLM for APSRs generation is challenging. **C1: Incorrect APSRs.** The first challenge is the incorrectly generated APSRs by LLMs possibly due to hallucination [13]. For example, when prompting one of the LLMs with the Prompt *"When using the standard library function "`free`" in C, what are the API parameter security rules the caller needs to follow to prevent security issues?"*, the LLM's output says: *"Before calling `free`, always check if the pointer is NULL."*. However, this response is incorrect because the `free` function does not pose a security risk when its argument is NULL. Our analysis shows that the accuracy of directly using LLM for APSRs generation is merely 11.9% (Section V-E), which needs to be addressed for precisely APSRs generation. What's more, verifying correct APSRs also poses difficulties, which may result from missing correct references for comparison and preventing the existing evaluation methods (e.g., BLEU [14]). Therefore, how to generate correct APSRs including the verification remains the first challenge in need of addressing. **C2: Overly-general APSRs.** The second challenge is the generated APSRs might be overly general, which is hard to be interpreted into accurate APSRs and applicable detection codes for API misuse detection. For example, Figure 2 illustrates one piece of APSR generated by LLM, saying "*the `ppdb` must be valid*", which is too general. Specifically, *"valid"* could be variously interpreted by different users as "not NULL", "requiring initialization", and "not being arbitrary values". However, only the explanation *"the `ppdb` must not be NULL"* is correct which could be precisely interpreted into applicable detection code. Existing approaches have shown effectiveness in using Word Sense Disambiguation (WSD) [15], [16] technique to eliminate natural language's ambiguity, while it fails to generate accurate APSRs due to its difficulties in combination with security knowledge. Since incorrectly interpreting the APSRs introduces false positives and false negatives in API misuse detection, proposing an approach to generate accurate APSRs to eliminate the over-generalization is an urgent need.

Our work. In this paper, we proposed GPTAid (short for Generating API Parameter securiTy rules from API source code for API mIsuse Detection) – a tool for automatically generating accurate and concrete APSRs with LLM and detecting API misuse caused by incorrect parameter use. GPTAid addresses the aforementioned challenges based on several observations. Since the API source code is the actual implementation of APIs, GPTAid first prompts to generate the raw APSRs based on the API source code. To verify the correctness of the raw APSRs, GPTAid adopts an execution feedback-checking approach to validate the LLM-generated APSRs. More specifically, we observe that violations of APSRs often lead to security-related API misuse and most of them result in runtime errors which can be caught by the monitoring tools (e.g., sanitizer). For example, Figure 1 describes one piece of APSR specifying *"parameter 2 must be released after calling `sqlite3_open`"*. The code snippet shows a misuse caught by the sanitizer in a real application which results in a memory leak. Based on this observation, we separate the validation process into three parts: Right code (C_r) generation, Violation code (C_v) generation and correct APSRs verification. Particularly, GPTAid proposes to generate the (C_r) based on a step-by-step approach with the API source code. To ensure accuracy, GPTAid monitors execution outputs and applies an LLM-empowered automatic program repair method to automatically repair the erroneous code. Then, GPTAid generates the (C_v) which violates the raw APSRs by modifying the (C_r). To make sure the (C_v) correctly violates the APSRs, GPTAid applies program repair approach which is also adopted in the previous step. Besides, a deeper analysis is conducted to filter out the incorrect (C_v), including applying the off-the-shelf static analyzer to locate API calls and parameters, and further checking the inconsistency between them (Section III-C). At last, GPTAid dynamically executes the (C_v) assisted by sanitizers, which output runtime errors for correct APSRs and success for incorrect APSRs.

To solve the overly-general APSRs, we propose to adopt code differential analysis to generate concrete APSRs, as the

programming language, being a formal language, describes the API more specifically than natural language. Take the APSR *"Parameter 2: The ppDb parameter must be a valid pointer to an sqlite3 structure before calling sqlite3_open,"* as an example. The calling code that violates this piece of APSR is `sqlite3_open(..., NULL)`, which sets the second parameter to `NULL` and exactly violates the APSR *"...must be a valid pointer..."*. Contrary to natural language, the violation code merely leads to one interpretation resulting in no confusion for detection code generation from APSRs. Leveraging this observation, we propose to analyze the modification operations between (C_r) and (C_v) to generate concrete APSRs. However, a large amount of redundant information exists in the code, which hinders the modified operation identification and contributes to the arising of inaccurate APSRs. To diminish the redundant code, GPTAid instructs LLM to identify the shared modified operation among different violation code that leads to the same runtime error. According to that, GPTAid acquires the concrete APSRs for each key modification that causes the API-related runtime errors with LLM (Section III-D).

We randomly selected 200 APIs from eight widely used libraries, including OpenSSL [17], SQLite [18], libpcap [19], libxml2 [20], libevent [21], libzip [22], zlib [23] and libcurl [24], and constructed a new dataset by analyzing documentation and API source code. We evaluated the effectiveness of GPTAid on APSRs generation on this dataset and the results show that GPTAid achieves a precision of 92.3% and a recall of 71.0%. GPTAid identifies eight distinct types of APSRs, which surpasses the performance of previous work [1], [2] with two more rule categories identified (Section V-B). GPTAid outperforms the existing state-of-the-art tools, such as Advance [6]), which generates only one-seventh as many APSRs as GPTAid due to incomplete documentation (Section V-D). In total, GPTAid found 210 unknown API misuses from 47 applications integrating on eight libraries, of which 150 have been confirmed by the application developers through our ethical reports. All the misuses are security-relevant and can lead to system crashes and Denial-of-service (DoS). We plan to open-source our code and data later for future research[1].

Contributions. We summarize the contributions as follows:

• **Novel technique.** We proposed a new approach to automate the APSRs generation using LLM. Our approach addresses two key challenges in directly using LLMs: the incorrect and overly-general APSRs arising. To solve these challenges, we adopt an execution feedback-checking approach to verify the correctness of the generated APSRs (Section III-C), and a code differential analysis is applied to generate the concrete APSRs with LLM (Section III-D). The generated APSRs are then proven to be effective through the application of API misuse detection, which outperforms the state-of-the-art detectors.

• **Insightful findings.** We implemented GPTAid on a subset of APIs from eight popular libraries. Among the generated 579 APSRs, we found 61.3% of them have no corresponding description in the documentation, which means our work helps enrich the documentation. We reported these APSRs to the library developers, and 76 of them have been confirmed. In total, all of the generated APSRs help detect 210 unknown security bugs, which could lead to severe security issues (such as system crashes), and 150 of them have been confirmed by developers after our ethical reports.

• **Suggestions.** Through the analysis of the GPTAid's performance, we got the chance to provide suggestions for API developers to enhance documentation for preventing API misuse (Section VI-D). Besides, suggestions on prompt design to simplify the process and minimize errors generated by LLM are also provided in Section VI-C.

[1]https://github.com/icy17/GPTAid/

II. Background

A. API Parameter Security Rules

APIs are functions provided by libraries that other software can call directly, reducing repeated implementations and easing development [25]. They are widely used in software development, and libraries typically offer API lists and documentation. In our research, we use LLM to generate rules for third-party library APIs. API Parameter Security Rules (APSRs) define constraints on API parameters, addressing constraints on both parameter values and operations. *(1) Parameter values*: API parameters serve as inputs for various operations within API, demanding that developers ensure compliance with specified values, such as the parameter must not be negative, the value of parameter 1 must not be larger than the size of parameter 2, and the member of the parameter must not be NULL. *(2) Operations on Parameters*: APIs can be used in complex call contexts, and the status of an API's parameters might be affected by multiple APIs. We define these types of rules as constraints on operations, such as the parameter must be freed later, the parameter must not be freed before, and so on. Violation of APSRs can result in security issues, such as crashes, memory corruption and denial of service. Therefore, detecting parameter-related API misuse is crucial for security. However, many APIs lack proper documentation, and creating APSRs manually is often hindered by limited expert knowledge and is time-consuming. Consequently, detecting parameter-related API misuses becomes a challenging task. In our research, we use LLM to generate various types of APSRs, such as value-related constraints, constraints among parameters, and constraints on operations. By applying these APSRs, GPTAid can detect bugs caused by incorrect parameter use. For example, GPTAid can detect null pointer reference (caused by incorrect parameter values), buffer overflow (caused by incorrect relationships among parameters), and memory leak (caused by incorrect operations on parameters).

B. Large Language Model

Large Language Models (LLMs), like OpenAI's GPT-3 with 175 billion parameters [26], are neural networks trained on vast datasets. This extensive training allows LLMs to perform

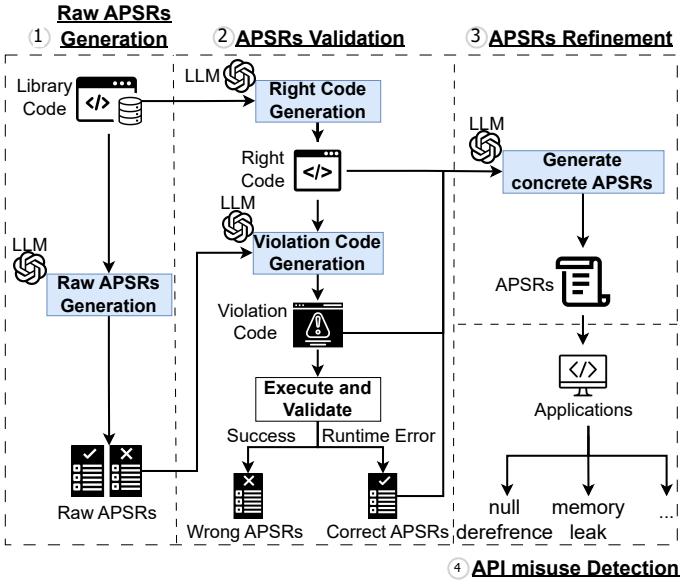

Fig. 3: Architecture of GPTAid

specific tasks without fine-tuning. LLMs like GPT interact with users through prompts, which are user-provided inputs describing the task to be accomplished. To enhance the reasoning ability of LLMs on various specific tasks, existing research has proposed some approaches for designing prompts (known as *prompt engineering*). Zero-shot prompts [27] do not include interaction examples. Few-shot prompts [28] guide LLMs by incorporating examples. The Chain-of-Thought [29] approach represents the state-of-the-art technique, enhancing reasoning through step-by-step design. Previous approaches [30], [31], [32] used LLMs for software security. In our research, LLM is employed to generate APSRs. However, the outputs of LLMs can be incorrect or overly general, which can not be easily resolved by mere prompt engineering. Identifying these incorrect or general responses of LLMs is quite challenging.

III. Methodology

In this section, we introduce GPTAid, which is designed for automatically generating accurate and concrete API Parameter Security Rules (APSRs) using LLM and detecting parameter-related API misuses. We start with the overview and use an example to illustrate the workflow of GPTAid, followed by a detailed description of each component.

A. Overview

Architecture. Figure 3 illustrates the architecture of GPTAid, consisting of four stages: Raw APSRs Generation, APSRs Validation, APSRs Refinement and API misuse detection. In Raw APSRs Generation (stage-1), GPTAid automatically constructs a prompt with the API source code and prompts LLM. This enables LLM to analyze the API source code and generate raw APSRs. However, the generated raw APSRs might be incorrect. Therefore, for each raw APSR, GPTAid validates the correctness of it based on execution feedback (stage-2). In this stage, GPTAid first instructs LLM to generate

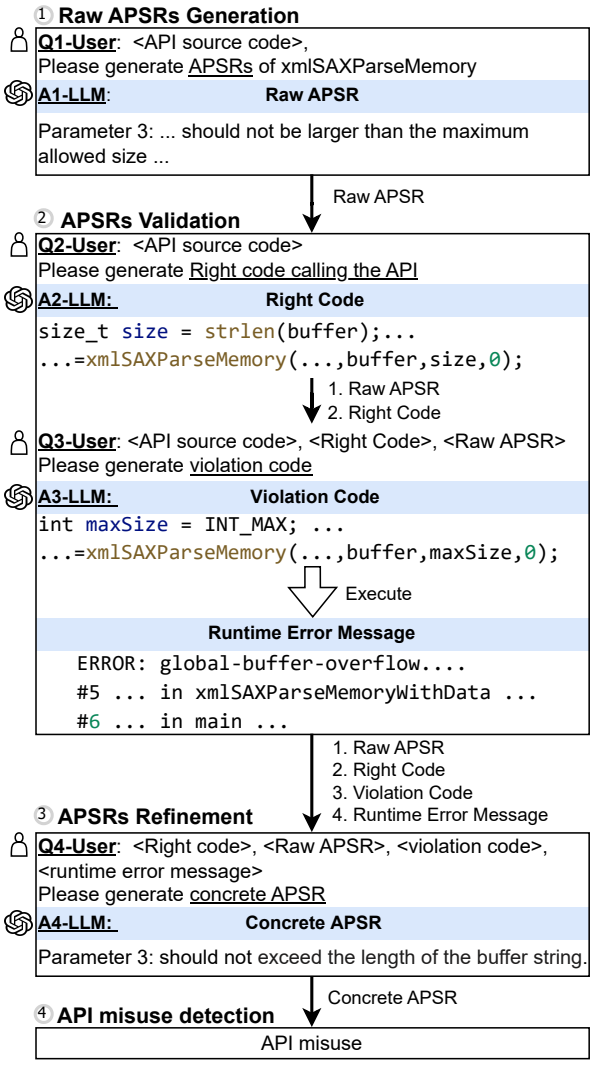

Fig. 4: An example of GPTAid's workflow

the right code (C_r) calling the target API. To ensure C_r executes without triggering any runtime error, GPTAid monitors execution feedback and automatically repairs runtime errors. Subsequently, for each raw APSR, GPTAid automatically instructs LLM to modify the C_r to generate the violation code (C_v) that violates the target raw APSR. GPTAid then automatically executes the C_v and monitors if a runtime error occurs to identify the correct raw APSRs. To enable the use of APSRs for API misuse detection, GPTAid refines all the correct APSRs to generate concrete APSRs (stage-3). For this purpose, GPTAid instructs LLM to analyze the difference of C_r and C_v to generate APSRs related to the modification operations. The APSRs generated are used to detect API misuse with CodeQL [33] (stage-4).

Example. The example in Figure 4 introduces the workflow of GPTAid. It describes the entire process of GPTAid generating the APSRs of API `xmlSAXParseMemory`, an API in the libxml2 library that "*parse an XML in-memory block and use the given SAX function block to handle the parsing callback*" [34]. First, GPTAid provides a prompt containing source code of `xmlSAXParseMemory` to LLM and LLM

User:
[source code of `pcap_dump`]
Please analyze the source code of the API `pcap_dump` and all the information provided before it. Then, respond with the following tasks:
Task 1: Based on the analysis of the source code, provide a concise description of the **functionality of the function**.
Task 2: **Locate the code** related to the Parameter 1 `user`.
Task 3: Identify the **security rules** that should be observed for the Parameter 1 to prevent misuse of the API `pcap_dump`. For each rule, include **code snippets that demonstrate a violation of the rule**...

LLM:
1. **Rule**: The `user` parameter should be validated to ensure it is a valid file handle and not NULL before using it.
Violation Code:

```
FILE* f = NULL;
pcap_dump((u_char*)f, &h, sp);
```

Fig. 5: Prompt example of Raw APSRs Generation

generates a raw APSR: *"Parameter 3: The size parameter should not be larger than the maximum allowed size to prevent denial-of-service attacks or memory exhaustion."* as is shown in *A1*. Then GPTAid instructs LLM to generate the C_r and then modify it to generate C_v that violates the raw APSR. To generate the violation code, LLM modifies the third parameter from `strlen(buffer)` to `INT_MAX`, shown as the C_r and C_v in *A2* and *A3*, respectively. Subsequently, GPTAid executes the C_v and catches a runtime error using the sanitizer [35], confirming the correctness of this APSR. GPTAid then instructs LLM to analyze the key modification operations by identifying the difference between the C_r and C_v and generate a concrete APSR that describes a relation between the values of parameter 2 and parameter 3, as shown in *A4*. Finally, the concrete APSR is used to detect API misuse.

B. Raw APSRs Generation

This stage aims to generate APSRs by analyzing API source code. Most of the existing work [4], [8], [1], [2] is limited to detecting specific types of API misuse, relying on specific code analysis rules for source code analysis. Manually constructing these code analysis rules is very time-consuming and limited by expert knowledge. LLMs are powerful tools that have the ability to comprehend code without the need of code analysis rules. Therefore, GPTAid uses LLM to generate APSRs by analyzing API source code automatically. The APSRs generated in this stage are referred to as *raw APSRs*.

Initially, instructing LLM to generate APSRs by directly analyzing the source code of APIs may seem straightforward. However, this simple approach dose not work well when dealing with intricate API source code that involves multiple parameters and complex implementation logic. Analyzing multiple targets from a large amount of code is challenging for LLM, causing missing rules. For example, when we provide LLM with the source code of `pcap_dump` – an API with three parameters – and instruct it to generate all related APSRs, it generates only one correct APSR, missing four others. To address this issue, we propose a method that break down the task. To break down the task, GPTAid begins by identifying the numbers and names of parameters of target API through static analysis. GPTAid then instructs LLM to

```
const char *filename="non_existing.db";
... = sqlite3_open(filename, NULL);  ← invalid!
```

Fig. 6: False violation code leading to unrelated API misuse

generate raw APSRs for each parameter rather than generating raw APSRs for all the parameters. This enables LLM to focus on one parameter at a time and simplifies the process of linking APSRs with their related parameters.

Prompt Design. To fully utilize LLM for generating raw APSRs, we evaluated three prompt engineering methods: zero-shot, few-shot, and Chain-Of-Thought. Our analysis showed that Chain-Of-Thought performed best, so we adopted it for prompt design. Detailed comparisons of them are provided in Appendix IX-B. The prompt consists of two parts: the API source code as input information and step-by-step instructions for the LLM to complete, as shown in Figure 5. We design each step's instruction based on the principle that the LLM can complete the step by analyzing the information given in the prompt, and that completing the step will aid the LLM in generating the APSRs. The instruction is divided into the following three steps:

❶ **Summarizing API functionalities** helps LLM to mine potential APSRs. There is a relationship between API functionalities and its potential APSRs. For example, when an API has memory allocation functionality, the corresponding APSR is the need to release allocated memory.

❷ **Locating parameter-related lines of API source code** reduces the amount of code that LLM needs to analyze. This step helps the LLM reduce the impact of irrelevant code on the correctness of generating the APSRs.

❸ **Generating raw APSRs with their violation code examples** can express the raw APSRs in a straightforward way. As mentioned earlier, the raw APSRs generated by LLM are overly general causing the gap between raw APSRs and downstream tasks. For example, as previously mentioned, the general APSR is: *"Parameter 2: The `ppDb` parameter must be a valid pointer to an `sqlite3` structure before calling `sqlite3_open`"*. There are three possible interpretations of *valid* in this APSR, but only one is correct. Overly general APSRs can result in errors during subsequent use. To solve this problem, it is essential to identify information that can represent a specific constraint of the API and use it as the supplementary information to the raw APSRs. Programming language is a formal language that is more concrete compared to natural language. Intuitively, for different APSRs, the code violating these APSRs is distinct and the violation pattern may provide insights into the constraints of the API. Based on this assumption, a violation code example can be used to represent a specific constraint of API. Therefore, in the last step of the instruction, we instruct LLM to generate raw APSRs with their violation code examples, providing supplementary information for these raw APSRs. In our evaluation, GPTAid achieved a recall of 84.4% in generating raw APSRs, generating more APSRs than other approaches (Section V-D).

C. APSRs Validation

As mentioned earlier, LLM may produce incorrect answers. This stage aims to validate the correctness of the raw APSRs generated by LLM in the previous stage, which contains challenges. The raw APSRs are expressed in natural language, and automating the assessment of their correctness can be quite challenging due to the absence of definitive reference information. After analyzing the reported API misuse in prior studies [3], [4], [5], [6], we identified that 94% of them lead to runtime errors during execution which can be caught by monitoring tools (Table VIII in Appendix). Therefore, we assume that if a raw APSR is correct, the code that violates the raw APSR will cause runtime errors. Based on this assumption, GPTAid validates the correctness of a raw APSR by analyzing the execution result of the code violates it. In a few cases, violating an APSR does not lead to runtime errors, causing GPTAid to fail in generating the corresponding APSR. We will discuss this limitation in Section VI-E.

Directly generating API calling code that violates the raw APSRs might lead to API misuse unrelated to the raw APSRs. Failing to differentiate whether API misuse results from a violation of the raw APSR can lead to the execution-based analysis incorrectly confirming an incorrect raw APSR as correct. For example, if LLM is instructed to directly generate code that violates the raw APSR of `sqlite3_open`: *"Parameter 1: The filename should be validated to ensure it refers to a legitimate, existing database file"*, LLM might generate code that looks like the code in Figure 6. While this code does violate the APSR successfully, it passes the second parameter as NULL which leads to a runtime error unrelated to this raw APSR. Based on the assumption, this raw APSR is categorized as correct, which leads to an incorrect APSR. It is challenging to distinguish these incorrect violations due to the absence of the correct API usage patterns. To solve this problem, GPTAid first generates the right code (C_r) calling the API without runtime errors and modifies the C_r according to raw APSRs to get the violation code (C_v) that violates the raw APSRs. Through the modification, GPTAid ensures the runtime errors of C_v are related to this modification based on raw APSRs. Finally, GPTAid verifies the correctness of raw APSRs by executing the C_v and analyzing if there are runtime error messages (*REMs*) related to the target API. This stage consists of three steps: right code generation, violation code generation, and correct APSRs verification. We delve into the details of these steps below.

Right Code Generation. The aim of this step is to generate C_r. Existing approaches primarily mine the API usage patterns from extensive code corpora and subsequently generate API-calling code [36], [37]. However, these approaches have a low success rate in generating code, and they often produce redundant code. Complex code can make compiling and running it successfully a challenge, while too much irrelevant code can complicate result analysis. The C_r should be as simple as possible to minimize the introduction of API-unrelated factors. Because LLMs' capacity to generate API calling code [30], supported by its vast knowledge of code, GPTAid utilizes LLM for generating C_r that calls API.

Example Right Code Generation Prompt Input
Given the following function [API_Name] in library [Lib_Name], Complete the following tasks. [API Source Code] Task1: Analyze the function code to find the invocation specification that the caller needs to follow correctly. Task2: Generate a complete code that calls the function in Linux. The code needs to meet the following requirements:...
Example Right Code Generation LLM output
[Right Code Calling API]
Example Automated program repair Prompt Input
Run result of the code is: [Runtime Error Messages] Please fix this code based on the run result. Please Follow the instruction in the first session! Note: I am using the program automation to run the code you gave, so please generate the code directly that will run correctly.

Fig. 7: Prompt example of Right Code Generation

To generate C_r, we design a prompt consisting of two parts: the API source code and step-by-step instructions. The step-by-step instruction contains two tasks: analyzing specifications for calling the API correctly based on the API source code and generating the C_r. Since correctly calling certain APIs requires constructing complex contexts, it is challenging for LLM to generate API calling code that satisfies these contexts accurately. To address this, GPTAid instructs LLM to perform automated program repair when compile errors or runtime errors arise. In this way, GPTAid can generate the correct calling code by modifying the incorrect context and gradually satisfying the complex context step-by-step. Specifically, GPTAid provides LLM with the session history that is used to generate the C_r, with any error feedback from execution and instructs LLM to fix code errors. When the execution of C_r is successful or the automated repair reaches the maximum repair times, the automated repair process stops. The prompt example of Right Code Generation is shown in Figure 7. Our study shows the effectiveness of GPTAid, achieving a success rate of 93.5% in generating the correct API calling code.

Violation Code Generation. The aim of this step is to generate the violation code (C_v) that violates raw APSRs by modifying the C_r based on the provided raw APSRs. To achieve this, GPTAid instructs LLM with a prompt containing a task description and four parts of information including the C_r, API declaration, raw APSR, and violation code example for raw APSR. The API declaration aids the LLM in analyzing the APSR and identifying the target parameter described within. The violation code example helps to provide specific details and clarify the APSR. By following the instructions, GPTAid can modify C_r to C_v in various ways, such as changing parameter values or altering the API calls in the calling context involving the same variable. The prompt template is shown in Figure 8. Similar to right code generation, C_v can contain errors and cannot run. We use the same repair method to solve this problem. As previously mentioned, the output

Input Information

1. Declaration of API 2. Right Code of API
3. raw APSR to be violated 4. Violation code Example

Input Instruction

Based on the above information,
please **modify the right code to violate** this rule:
[raw APSR]
Here is an example of a code that violates the rule;
analyze the code to determine how to violate the rule by modifying the right code...(skip explanation)
violation code example:
[Violation Code Example]

Fig. 8: Prompt template of Violation Code Generation

generated by LLM may contain errors. In this step, the error could be that LLM may not correctly modify the C_r according to the instructions leading to the APSR-unrelated modification.

To address this, GPTAid roughly checks whether the modifications are wrong by automatically analyzing whether the modifications of the code are consistent with what the raw APSRs describe. GPTAid first automatically identifies the differences between the C_r and the modified code. GPTAid then utilizes abstract syntax tree (AST) analysis to pinpoint target API calls and which parameter is associated with the modified code snippets. This process identifies the modified parameter (C_{para}) and the location relation (C_{loc}) between the modified code snippets and the target API. Meanwhile, GPTAid analyzes the raw APSRs using location keywords (e.g., before) to extract the location relation (R_{loc}) of the described action relative to the target API and gets the target parameter (R_{para}) directly from the Raw APSRs Generation stage. GPTAid assesses the correctness of the modification by comparing the consistency of (C_{para}, C_{loc}) and (R_{para}, R_{loc}).

Correct APSRs verification. This step aims to verify the correctness of the APSRs based on the runtime execution output of C_v. As previously mentioned, the assumption is that an APSR is considered correct if the C_v leads to runtime errors during execution. However, runtime errors might be unrelated to the target API, as shown in Figure 9, the C_r to call the API `sqlite3_bind_blob64` is in Figure 9(a), and the APSR to be checked is: *"Parameter 3 must not be NULL"*. To violate this APSR, LLM modifies the C_r to the C_v in Figure 9(b). This modification sets the variable `data` to NULL, precisely violating the APSR. As Figure 9(c) shows, the violation code causes a runtime error. However, the variable `data` is also passed to `strlen`. Unfortunately, passing a NULL parameter to the `strlen` function leads to a null pointer dereference, resulting in a runtime error unrelated to the API.

To distinguish between errors caused by the target API and those caused by unrelated factors, we design a process to analyze REMs generated by the monitors automatically. REMs describe where an error occurred in the code and provide a stack trace showing the call sequence leading to the error. First, GPTAid analyzes the error trace to determine if the error occurs at the location of the code where the target API is called. Then, to further determine if the error is API-related, GPTAid examines the subsequent call sequence in the error trace to see if the error occurs in the implementation of the target API. For example, in the Figure 9(c), GPTAid first identifies the error

```
const char *data = "example data";
...=sqlite3_bind_blob64(..., data, strlen(data), ...);
```

(a) right code

Generate violation code

```
const char *data = NULL;
...=sqlite3_bind_blob64(..., data, strlen(data), ...);
```

(b) violation code

Execute

```
==290442==ERROR: AddressSanitizer: ...
    #1 ... in __interceptor_strlen ...
    #2 ... in main test.c:49
```

(c) REM of violation code

Fig. 9: Violation code leading to an unrelated bug

```
BIO_free(bio);
bio = NULL;
// Attempting to use `bio` after modification
BIO_callback_fn_ex fn = BIO_get_callback_ex(bio);
```

Fig. 10: Modification contains multiple operations

in the `main` function (`main test.c:49`), where there is a line that includes function calls to `sqlite3_bind_blob64` and `strlen`. This indicates that the error is very likely caused by the API. GPTAid then analyzes the stack trace frames above `main` to confirm whether the error originated within the API rather than from `strlen`. The frame above is: `in __interceptor__strlen`, indicating that this REM is related to `strlen`, not the API. Based on the above analysis, GPTAid can ascertain whether the REM is related to the API.

D. APSRs Refinement

As previously mentioned, the APSRs generated by LLM are general, making them unsuitable for use as reliable information for API misuse detection. Therefore, refining APSRs to express concrete constraints becomes crucial. Considering code is more straightforward and concrete (Section III-B), we propose an approach based on code differential analysis to guide LLM in generating APSRs consistent with code. Specifically, the C_v is a modification of the C_r and the modification leads to API misuse. Well-defined APSRs should contain constraints aimed at preventing incorrect API usage introduced by these modifications. Based on this phenomenon, analyzing the modifications between the C_r and the C_v and generating APSRs that describe these modifications can be helpful in producing concrete APSRs. However, when analyzing code modifications involving multiple operations, LLM may struggle to identify the specific modification operations directly related to API misuse. This issue could lead to LLM generating an incorrect APSR based on the unrelated operations. For example, as shown in Figure 10, for API `BIO_get_callback_ex`, the generated C_v is shown in the figure, which results in an error. The difference between C_v and C_r is that C_v frees the `bio` parameter and then sets it to NULL before calling the target API. The error is caused by the passing NULL as the parameter, and the error is not related to the `free` function. In this case, identifying the exact operation is challenging due to the multiple modifications involving both the `free` function and the NULL pointer.

Information in prompt

Info 0. Declaration of API ***Info 1***. Right Code ***Info 2***. Violation Code
Info 3. Runtime Error Messages ***Info 4***. Raw APSR

step-by-step instructions in prompt

1. Enumerate the **key differences** between the correct and error API calls.
2. Determine the potential root cause of the error based on the **Info 3(REM)**.
3. Check if the **rule(Info 4) above is related to this error**.
4. For **each identified difference**, evaluate its probability of being the primary root cause of the error based on the provided error code and Result of the Run.
5. **Rank** these causes based on their likelihood.
6. For the most probable causes, **gesnerate specific security rules** to prevent such errors in the future...(skip some explanation)

Fig. 11: Prompt template of APSRs Refinement

To solve this problem, GPTAid first groups different violation code that lead to the same runtime error, and then identifies the shared operations as the key operations. Specifically, we observe that the same API-related runtime errors typically result from the same operations in the code. Therefore, GPTAid analyzing different C_v that leads to the same API-related runtime errors, aiming to identify common code modification operations shared among them. This helps GPTAid pinpoint the key operations causing the API misuse from several modification operations. For this purpose, GPTAid begins by grouping C_v that share the same REMs. To efficiently group REMs, GPTAid focuses on essential information within the REMs while discarding irrelevant factors, such as process IDs, which could distort clustering. Specifically, it identifies the potential causes described in the REMs and analyzes the associated error stack trace to identify the call sequence using the same method as in APSRs validation. Then, it groups REMs based on these crucial details. Subsequently, GPTAid provides LLM with information from each cluster, instructing LLM to analyze differences between all the C_v and their C_r to identify the modification operations in code. GPTAid then instructs LLM to analyze the common modifications operations among them to identify the key operations. Finally, GPTAid instructs LLM to generate the APSRs containing descriptions of the key operations for each cluster.

Prompt Design. To enable LLM to analyze all available information and generate concrete APSRs based on the key operation, we design a prompt consisting of information and step-by-step instructions that LLM needs to complete. There are six main steps of instructions: identify differences, analyze shared REMs, analyze raw APSRs, analyze differences, rank the possibilities and generate APSRs. The prompt template when only one C_v in a cluster is shown in Figure 11. Specifically, ❶ Identifying differences between each C_r and C_v pair in a group helps the LLM pinpoint all modification operations linked to the same runtime error. ❷ Analyzing shared REMs enables the LLM to identify potential causes of API misuse as described in the REMs. ❸ Analyzing raw APSRs enables the LLM to identify potential API misuse causes as described in the APSRs. This step is omitted for the cluster with multiple codes, as their distinct raw APSRs are less likely to directly relate to the root cause. ❹ Analyzing all the differences and identify one root cause helps LLM to identify key operations lead to the API misuse among numerous modification operations. ❺ Ranking the possibilities

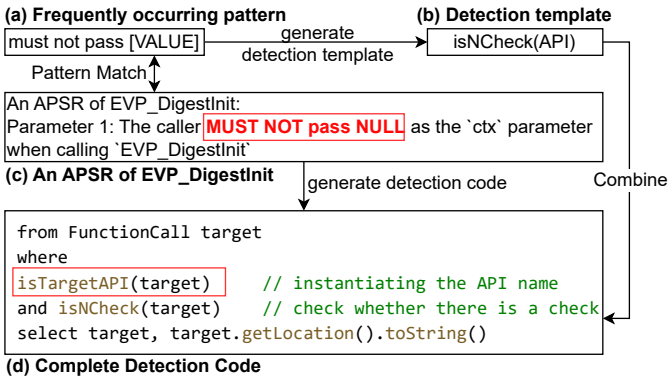

Fig. 12: An example of detection code generation

allows the LLM to evaluate all potential causes identified previously and determine the most likely cause of API misuse. ❻ In the generating APSRs step, GPTAid instructs LLM to generate concrete APSRs based on the operation and the most likely cause. In this way, APSRs are refined and can be used to generate detection rules for API misuse detection. Our experiment shows GPTAid achieves an accuracy of 92.3% in generating APSRs (Section V-E).

E. API misuse detection

In this stage, we use APSRs for API misuse detection. Our approach draw inspiration from Advance [6], which detects API misuse through security rules (referred to as IA in Advance) using CodeQL [33]. CodeQL is a tool for static analysis on target applications based on detection QL code composed using detection rules. Advance employs Natural Language Processing (NLP) techniques for clustering predicates in IA through frequent subtree mining, manually constructs detection code templates for each frequently occurring predicate, and finally automatically generates detection code for each IA by combining these templates. We start by manually identifying the frequently occurring description patterns across all APSRs and creating detection code templates for these patterns. Subsequently, GPTAid automatically combines these templates and instantiates the parameter index and API name to generate the complete detection code (QL code) and employs this code for API misuse detection using CodeQL. For example, the process of detection code generation is shown in Figure 12. Through our manual analysis of APSRs, we identify one of the frequently occurring description patterns is *"the caller must not pass [VALUE]"* (Figure 12(a)), where `VALUE` can be any value. We manually created a detection code template for this pattern (isNCheck function in Figure 12(b)). This function employs data/control flow analysis to ascertain whether the target parameter has been checked before the API is called. Figure 12(c) shows the APSR for API `EVP_DigestInit`, specifying that *"the caller must avoid passing NULL"*, which matches the identified pattern. GPTAid automatically combines the detection code template (isNCheck) and instantiates the API name and parameter index to generate the detection code, as depicted in Figure 12(d).

IV. Implementation

In this section, we detail the implementation of the components of GPTAid.

LLM settings. For LLM tasks, we utilize the state-of-the-art model gpt-3.5-turbo-0613 developed by OpenAI [26]. We conducted a temperature experiment to select the optimal temperature for maximizing the performance of the model in each task. Detailed information about this experiment is provided in Appendix IX-B. For raw APSRs generation, right code generation, violation code generation, and APSRs refinement tasks, the temperature of LLM is set to 0, 1, 0, and 1 respectively. All other model parameters are set to default, and we use the zero-shot approach to design prompts.

Preprocess. We first crawl the API lists from the libraries' official websites. We then use Tree-sitter [38] to parse the Abstract Syntax Trees (ASTs) of the code. This allowed us to extract API source code from the library's source code. Furthermore, for each library, we manually prepared the required header files, necessary files for the APIs, and compilation options in advance to help LLM generate the API calling code. These are one-time tasks for a library and taking less than an hour for completion by an individual.

APSRs Generation. When generating violation code, we use Tree-sitter [38] to analyze the ASTs of code. While executing the code, we use two monitoring tools, ASAN [35] and Valgrind [39], to capture memory-related runtime errors. We chose these tools because they are highly popular and provide comprehensive monitoring of potential issues during execution. The maximum number of automated program repair attempts is set at 10 for Right Code Generation and 5 for Violation Code Generation. All the complete prompts used by GPTAid are available online [2].

API Misuse Detection. We cluster the APSRs generated by GPTAid and generate detection code templates for five frequently occurring patterns, including *[API-A]* must not be called before *[API-B]*, *[API-A]* must be called after *[API-B]*, the caller must not pass *[VALUE]*, the parameter must not be used later, and parameter must be initialized. By applying these APSRs, GPTAid can detect bugs such as memory leak, NULL pointer dereference, double free, and so on. To ensure efficiency, we employ intra-procedural analysis for detection.

V. Evaluation

In this section, we evaluate the effectiveness of GPTAid in APSRs generation, API misuse detection, and individual components. We also compare GPTAid with state-of-the-art approaches [4], [6], [11]. Subsequently, we conduct an ablation study to show GPTAid's improvement on LLM, followed by an empirical analysis of the results and a case study.

A. Settings

Dataset. we utilized five datasets to evaluate the effectiveness of GPTAid:

[2]https://github.com/icy17/GPTAid/tree/main/prompt

TABLE I: Library Details

Library	Functionality	#API
libpcap	Network	71
libxml2	XML parser	1614
sqlite3	Database	294
openssl	Cryptography	5478
libevent	Event handling	401
libzip	File compression	120
zlib	Data compression	69
libcurl	Network Transfer	76
Total	**/**	**8123**

• *Corpora of library source code (C_{code}).* We selected eight widely-used libraries from different categories, including OpenSSL [17], SQLite3 [18], libpcap [19], libxml2 [20], libevent [21], libzip [22], zlib [23] and libcurl [24], based on their popularity on GitHub (measured by the number of stars) and their prevalence on Ubuntu. In total, we collected 8,123 APIs with 2.53M lines of code. Detailed information about these libraries is provided in Table I.

• *Ground-Truth dataset for APSRs Generation (D_{gt}).* To evaluate the effectiveness of APSRs Generation, we constructed a dataset by randomly selecting 25 APIs from all the APIs across each of the 8 libraries forming a total of 200 APIs, which intends to reduce the potential bias from frequently called APIs. To create a comprehensive GroundTruth (GT), we first verified the APSRs generated by Advance [6] and Goshawk [4], then analyzed the API source code and documentations to identify overlooked APSRs. Through our analysis, we generated 404 APSRs for 200 APIs in total.

• *Comparison dataset (D_{comp}).* To compare the effectiveness of GPTAid with previous studies, we constructed a dataset consisting of bugs and their APSRs. The bugs include those were sourced from the previous studies within our scope and those were detected by GPTAid. Since GPTAid is implemented in user space, we excluded Linux kernel bugs, identifying 86 bugs from Advance and 10 bugs from Goshawk. Since IPPO [11] does not disclose the locations of bugs, we only used the results from GPTAid, Goshawk [4]and Advance [6], which contained a total of 306 bugs and 58 APSRs.

• *Standard Dataset for API misuse detection (APIMU4C [40]).* APIMU4C is a standard dataset that focuses on API misuse. We utilized 12 bugs within our scope to evaluate the performance of GPTAid in detecting bugs.

• *Applications for API misuse Detection (D_{app}).* To evaluate the effectiveness on API misuse detection, we selected 10 popular applications for each library based on the popularity (reflected by the number of github stars) to form a total of 47 applications. All the applications have GitHub stars exceeding 1,000, indicating their widespread usage (details are shown in Table IX in the Appendix).

Platform. We conducted experiments on a 64-bit server running Ubuntu 18.04 with 16 cores (Intel (R) Xeon (R) CPU v4 @ 2.10GHz), 440GB memory, and 11TB hard drive.

TABLE II: Effectiveness of APSRs Generation

Library	Precision	Recall	F1	Cost Per API ($)
libpcap	0.96	0.78	0.86	0.1
libxml2	0.89	0.68	0.77	0.12
sqlite3	0.91	0.82	0.86	0.1
openssl	0.94	0.68	0.79	0.13
libevent	1.00	0.63	0.77	0.11
libzip	0.86	0.73	0.79	0.13
libcurl	0.95	0.64	0.76	0.12
zlib	0.88	0.73	0.80	0.11
Overall	0.92	0.71	0.80	0.12

```
  if (pcap_set_rfmon(handle, rfmon) != 0) {...}
  ...
  pcap_close(handle);
+ //Calling pcap_set_rfmon with a closed handle
+ pcap_set_rfmon(handle, rfmon);
  return 0;
```

Fig. 13: Example of a false positive in APSRs Generation

B. Effectiveness

In this section, we evaluate the effectiveness of GPTAid on APSRs generation and API misuse detection.

Effectiveness of APSRs Generation. We evaluated the effectiveness of GPTAid in generating APSRs on D_{gt} by manually analyzing their correctness. We used three metrics to show the results: precision, recall, and F1 score. With a cost of only $0.12 per API, GPTAid generates 311 APSRs with a precision of 92.3% and a recall of 71.0%, as shown in Table II.

We analyzed these 24 false positives (FPs) and identified a primary cause: 75% resulted from incorrect key operation identification during APSRs refinement. In this stage, LLM needs to identify the key operations by comparing correct and violation codes and generate the APSRs related to the key operations. However, in cases with intricate modifications, the LLM often misidentifies key operations, leading to false positives. For example, Figure 13 illustrates an incorrect APSR for `pcap_set_rfmon` generated by LLM. The figure highlights differences between correct and violation code, with lines starting with + indicating added code. The violation code passes a closed variable to `pcap_set_rfmon`, causing the target API to use the variable after it has been closed, leading to a crash. Consequently, the correct rule should be: *"Parameter 1 must not be closed before calling `pcap_set_rfmon`"*. However, LLM misinterprets the key operation as a violation code calling the API twice, and generates the APSR as *"Parameter 1 (p) should not be called with the `pcap_set_rfmon` function more than once"*, thereby generate an incorrect APSR.

We conducted an analysis of these 117 false negatives (FNs) and identified three primary underlying reasons. ❶ **Missing in Raw APSRs Generation** accounts for 49.6% of all FNs. Since GPTAid relies on LLM to generate APSRs, and if LLM fails to generate the APSRs during the initial step, it leads to APSRs being missing. ❷ **LLM's unexpected behavior**, contributing to 13.7% of all FNs. When generating violation code, despite GPTAid conducting a consistency check to identify the incorrect modifications, few unintended errors may go undetected, causing incorrect modifications. For example, a modification may modify the correct parameter in the correct location but with incorrect content. Identifying such errors becomes challenging due to the complexity of code analysis. The incorrect modifications cause LLM to deviate from violating the correct APSR, and generate a violation code without runtime errors. Consequently, GPTAid mistakenly marks this APSR as incorrect, causing false negatives. Additionally, similar to the reason for FPs mentioned earlier, the LLM may misidentify key operations, generating incorrect APSRs and missing the originally correct ones. ❸ **Failure to perform APSRs validation**. Failure to generate the right code to invoke the API was responsible for 13.7% of all FNs. Since we validate APSRs by executing the code, any inability to generate the right code for API calls prevents validation, resulting in all APSRs for that API being missed. By analyzing the results, we found that the success rate for generating right code is 93.5%. However, for the remaining 6.5% of APIs, all APSRs are missing.

TABLE III: APSRs type

Value					Action		Others
C_1*	C_2	C_3	C_4	C_5	C_6	C_7	
17	90	3	7	9	85	92	8

[1] C_n denotes category-n, detailed in Section V-B.

Analysis of generated APSRs. We utilized the classification approach for APSRs from prior work [1], [2] to analyze and categorize all the APSRs generated by GPTAid on D_{gt} into eight categories, as detailed in Table III. This includes five previously identified categories and two new action-related types. These eight categories of APSRs are as follows:

❶ **Range.** This category defines constraints to avoid invalid ranges of parameter values. For example, a simplified APSR for the API `adler32` is *"len must not be a negative value"*.
❷ **NULL.** This category specifies constraints to ensure parameter values are not NULL. For example, an APSR of `sqlite3_open` is *"ppDb parameter MUST NOT be null"*.
❸ **Member.** This category specifies constraints for values of parameter member. For example, an APSR of `pcap_dump` is *"fields of the struct must not all be -1"*. Unlike previous findings of APSRs in Java, C APIs' protected structs limit access to internal variables, resulting in fewer member APSRs.
❹ **Relation.** This category defines the constraints on the relationships between different parameters. For example, an APSR of `sqlite3_randomness` is *"pBuf must be allocated with a size equal to or greater than N"*.
❺ **Format.** This category describes constraints on parameter types or formats of content. For example, an APSR of API `sqlite3_open` is *"the filename must be null-terminated"*.
❻ **Action-Do.** This category of rules describes constraints on the actions required for the parameters. For example, an APSR of `zip_register_progress_callback` is *"The 'za' should be initialized by calling zip_open"*.
❼ **Action-Not Do.** This category specifies actions that are disallowed for parameters. For example, an APSR of `curl_share_cleanup` is *"must not be freed before"*.

TABLE IV: Detection results of GPTAid

Library	#APIs	#APSRs	#Bugs
libpcap	17	36	7
libxml2	30	19	5
sqlite3	94	136	7
openssl	163	206	51
libevent	47	48	140
libzip	27	49	0
zlib	22	43	0
libcurl	30	42	0
Total	431	579	210

TABLE V: Effectiveness of Individual Components

Libs	raw APSRs generation	APSRs validation		APSRs refinement
	Recall	Precision	Recall	Presicion
OpenSSL	0.85	0.94	0.84	0.93
SQLite3	0.89	0.96	0.82	0.93
libxml2	0.84	0.95	0.81	0.88
libpcap	0.95	0.95	0.87	0.94
libevent	0.84	0.97	0.69	0.9
libzip	0.82	0.96	0.78	0.88
zlib	0.77	0.98	0.88	0.91
libcurl	0.78	0.98	0.74	0.94
Total	**0.84**	**0.96**	**0.80**	**0.91**

❽ **Others.** This category includes APSRs that don't fit into the above categories. For example, an APSR of `gzflush` is *"The caller MUST NOT use `gzflush` with the flush parameter if there is a seek request pending"*.

Analysis of detected API misuse. We evaluated the effectiveness of GPTAid in API misuse detection using APSRs generated by GPTAid. We first analyzed the effectiveness of GPTAid on APIMU4C, where it successfully identified 11 out of 12 bugs. The missed bug is a lock-missing-unlock bug. GPTAid failed to detect it because GPTAid could not capture exceptions caused by the missing unlock during execution, thus preventing the generation of the necessary APSRs for bug detection. We then evaluated the ability of GPTAid to detect new bugs. To ensure cost-effectiveness, we selected a subset of APIs that were called more than 10 times within D_{app} for misuse detection, totaling 431 APIs. GPTAid generated 579 APSRs for this subset. Not every API has an APSR, as we did not apply additional filtering criteria. We applied CodeQL to detect API misuse within D_{app}, detecting a total of 210 unknown bugs, 150 of which have been confirmed by the developers after reports. Results of API misuse detection are shown in Table IV and the details of API misuse are shown in Table X in Appendix. The precision of GPTAid on bug detection is 77.2%, which is acceptable for static analysis-based detection, and it is higher than that of comparable tools like IPPO. Our analysis shows no false positives (FPs) from incorrect APSRs, as we only generate detection code template for frequently occurring rules, minimizing the impact of errors. One primary cause of false positives (FPs), accounting for 45.2% of all FPs, is the limitation of intra-procedural analysis. For example, GPTAid detects potential null pointer dereference by verifying the presence of a NULL check before calling the API. However, if this check is located in another function, the intra-procedural analysis may erroneously report a bug due to the absence of the check within the current function.

C. Evaluation of Individual Components

Effectiveness of Raw APSRs Generation. We evaluated the effectiveness of raw APSRs generation on 200 APIs in D_{gt}, finding GPTAid generated 2858 raw APSRs with a recall of 84.4%, as shown in Table V. This stage relies on the LLM's capabilities, and the occurrence of false negatives (FNs) is also linked to these capabilities. We attempted to identify the causes of these FNs by examining the API source code. One possible reason for FNs is that the LLM generates APSRs from source code analysis, focusing on the functionality of API parameters. Occasionally, this results in stereotypical inferences, contributing to FNs. For example, when a parameter serves to pass a file name, the LLM tends to generate rules focused on file names, such as: *"the filename must be validated to ensure it does not contain any path traversal"*. However, stereotypical inferences may limit the LLM's code analysis, causing it to overlook other rules which is unrelated to the functionality. This accounts for 44% of all FNs.

Effectiveness of APSRs Validation. In this section, we calculate the following: True Positives (TP), which are APSRs that are successfully and correctly validated; False Positives (FP), which are APSRs that are successfully but incorrectly validated; and False Negatives (FN), which are APSRs that failed to validate. We then use recall ($\frac{TP}{TP+FN}$) and precision ($\frac{TP}{TP+FP}$) to evaluate the effectiveness of GPTAid in APSRs validation. Based on our analysis, GPTAid achieved a precision of 96% and a recall of 80.2% in APSRs validation.

We analyzed all FNs and identified a major cause. The LLM struggles to generate right code and violation code, resulting in 75% FNs. For example, a raw APSR of `sqlite3_bind_int` is: *"Parameter 3: Prevent the use of the 'iValue' parameter in a multi-threaded environment without proper synchronization"*. To generate the violation code, LLM must generate a multi-threaded call that violates this specific rule, which is a complex task. LLM struggles to generate that complex violation code that can be compiled, leading to GPTAid's inability to validate this APSR.

We analyzed the results and found a major cause of FPs: errors introduced by LLM during the modification. This accounts for 52% of all FPs. When LLM tries to violate an APSR, it may introduce errors or not fully comply with the APSR's requirements for violation. For example, a raw APSR of API `sqlite3_vtab_in_frist` is *"Parameter 1: The pVal parameter should not be NULL"*. When LLM tries to violate the rule, it inappropriately alters the initialization of the parameter 2, turning it into an invalid variable and causing a runtime error. This unrelated error leads to the APSR being mistakenly identified as correct. Although GPTAid conducts coarse checks on violation modification correctness, a few unexpected behaviors still arise.

Effectiveness of APSRs Refinement. In this section, we assess the effectiveness of APSRs refinement by measuring the proportion of APSRs that are accurately transformed into

TABLE VI: Comparison with State-of-the-Art tools.

	GPTAid	Advance	IPPO	Goshawk	D_{comp}
#APSRs	53	7	/	8	58
#Bugs	243	99	0	10	306

[1] / means we cannot calculate this result.

downstream rules (precision). We categorize refinements as incorrect if they result in general rules or deviate from the original violation. Based on our analysis, GPTAid achieves an accuracy of 91.5%. We analyzed the incorrect refinement results and identified a primary error source: LLM identifies incorrect key modification operations. In few cases of violation code with complex modifications, LLM struggles to select the key operation from among multiple code snippets, resulting in inaccurate APSRs summaries. For example, in the context of multi-threaded APSRs, modifications include numerous multi-threading-related operations and key operations that cause runtime errors. LLM generates incorrect multi-threading-related APSR based on these operations.

D. Compare to the State-of-the-Art

In this section, we compared the effectiveness of GPTAid with three state-of-the-art tools: Advance [6], IPPO [11], and Goshawk [4]. These tools were selected from the categories summarized in Section I (analyzing API calling code, documentation, and API source code) and were frequently referenced in previous studies. We evaluated the effectiveness of these three tools and GPTAid on D_{comp} for APSRs generation and API misuse detection, based on the number of APSRs generated and bugs detected. The results of the comparative experiments are shown in Table VI.

Advance. Advance extracts APSRs from documentation by identifying strong sentiments in documents and utilizes these APSRs to detect API misuse. We compared its effectiveness with GPTAid in both APSRs generation and API misuse detection. We consider all APSRs described in the documentation as those generated by Advance and count the associated API misuses as bugs found by Advance. We evaluated Advance on D_{comp} and found that Advance can extract a maximum of 7 APSRs and identify up to 99 API misuse. In comparison, GPTAid generated 6 times more APSRs. To understand the unexpectedly results of Advance, we thoroughly analyzed the results and documentations and identified two possible reasons for the missing APSRs. ❶ Lack of Explicit Description in API Documentations. Through our analysis of the API documentations, we found that 88.0% of the APSRs in D_{comp} are not explicitly written in the documentation. These rules are often considered as common knowledge among developers and are therefore omitted. ❷ Descriptions without Strong Sentiment. Advance extracts IAs by recognizing strong sentiment in documentations. However, without strong sentiment in APSR descriptions, Advance cannot extract corresponding information, leading to missed APSRs. GPTAid missed 2 APSRs and 58 bugs detected by Advance because its monitors couldn't catch the exceptions during dynamic execution. Despite this, GPTAid identified more APSRs and bugs, showing that it performs better than the documentation-based method.

TABLE VII: Results of ablation study

	FP	TP	Precision	F1 Score
LLM	2517	341	0.12	0.21
LLM+S_v	74	284	0.79	0.74
LLM+S_v+S_r	24	287	0.92	0.80

[1] LLM means directly using LLM to generate APSRs.
[2] S_v is APSRs Validation. S_r is APSRs Refinement.

IPPO. IPPO identifies bugs by identifying inconsistent security operations within path-pairs in the API calling code. We applied IPPO on D_{comp} for bug detection and compared the results with those obtained using GPTAid. The results showed that IPPO did not detect any bugs. By analyzing the results, we identified two main reasons for the missing bugs: ❶ IPPO faces challenges in identifying security operations. IPPO detects bugs by comparing path pairs for inconsistent security operations but struggles to determine if an API is a security operation due to its limited understanding of APIs. This limitation can result in missed bugs. ❷ IPPO cannot detect bugs where inconsistent security operations are absent. IPPO identifies bugs by finding inconsistent security operations in path pairs. However, if API misuse occurs due to the absence of a security operation in all paths, IPPO cannot detect it because there is no inconsistency.

Goshawk. Goshawk can identify APIs in libraries that have allocation and deallocation functionality and detect use-after-free and double-free bugs based on the identified APIs. Goshawk can extract 8 APSRs and detect up to 10 bugs on D_{comp}, compared to the 53 APSRs and 243 bugs identified by GPTAid. By analyzing the results, we identified two primary reasons for the FNs in Goshawk: ❶ Focusing on the allocation/deallocation presents limitations. As previously mentioned, Goshawk's rule extraction from library code is constrained by predefined analysis rules, preventing it from extracting other types of rules. Consequently, Goshawk can only extract allocation/deallocation APSRs and cannot detect other types of bugs. Allocation/deallocation-related APSRs comprise only 25.9% of the APSRs in D_{comp}. ❷ Filtering functions based on name analysis results in missing. Goshawk leverages Natural Language Processing (NLP) techniques to analyze function names before conducting static analysis on library functions. This process filters out functions whose names lack relevance to allocation/deallocation operations, which may incorrectly exclude relevant APIs with less obvious names, leading to false negatives. GPTAid missed 3 APSRs and 5 bugs detected by Goshawk. The missed APSRs are caused by "Missing in Raw APSRs Generation" and "Failure to perform validation" (details in Section V-B). The missed bugs primarily occurred because GPTAid only performs intra-procedural analysis, missing bugs that require inter-procedural analysis. The results show that GPTAid generates more diverse APSRs and detects more bugs than Goshawk.

E. Ablation Study

In this section, we evaluated the contribution of GPTAid to the enhancement of LLM. We evaluated the effectiveness of directly using LLM, LLM+APSRs validation (LLM+S_v), and

LLM+APSRs validation+APSRs refinement (LLM+S_v+S_r). We refer to the APSRs generated directly using LLM as $APSR_1$, the APSRs generated by LLM+S_v as $APSR_2$, and the APSRs obtained by LLM+S_v+S_r as $APSR_3$. We calculate precision and F1 score as evaluation metrics.

Contribution of APSRs Validation. In this section, we evaluated the impact of APSRs validation on enhancing LLM accuracy. We conducted a comparison between the $APSR_1$ obtained directly using LLM and the $APSR_2$ obtained after APSRs validation, and the results are presented in Table VII. The findings reveal a significant improvement in precision, increasing from 0.12 to 0.79 and the F1 score increasing from 0.21 to 0.74. This improvement shows the effectiveness of our method in validating the correctness of APSRs.

Contribution of APSRs Refinement. In this section, we evaluated the impact of APSRs refinement on generating concrete APSRs. We compared the $APSR_2$ with the $APSR_3$. The results are presented in Table VII. The results indicated an improvement in both precision and f1 score. Precision has increased from 0.79 to 0.92, and the F1 score has risen from 0.74 to 0.80. This suggests that the APSRs refinement stage generate concrete APSRs.

F. Findings

Document errors leads to API misuse. We discovered that APIs whose code examples in documents are incorrect are more likely to result in API misuse. Based on our analysis of the results of API misuse detection, we observed that the `EVP_DigestInit_ex` API exhibited the highest percentage of misuse in the applications. After analyzing the documentation, we discovered that the API documentation [41] includes example code for using the `EVP_DigestInit_ex` API. However, the example fails to check if the parameter 1 is NULL, posing a potential security risk of null pointer dereference. The API misuse in the documentation may influence users to adopt a similar usage when calling this API.

APSRs enhance the quality of documentation. We generated 579 APSRs for 431 APIs, improving the quality of their documentation. We discovered that 61.3% of the APSRs we identified lacked explicit descriptions in the documentation. This makes it challenging for users to learn the APSRs they must follow from the documentation, consequently increasing the risk of API misuse. Existing approaches in rule extraction through documentation analysis, such as Advance [6], face limitations due to the lack of security descriptions in the documentation. Our approach can automate the generation of APSRs to enhance the security descriptions in the documentation. We reported these missed APSRs to the developers of the libraries, and 76 of them were confirmed. Some developers considered these APSRs common sense and didn't include them in the documentation. However, without clear guidelines, developers may accidentally violate these APSRs, leading to security issues. Additionally, relying solely on common sense for bug detection can cause issues, as some APIs have internal checks. This means violations may not cause security issues, leading to false positives in bug detection.

```
1 /* php-src/ext/openssl/openssl.c   */
2 //APSR of EVP_DigestInit: Parameter 1 must not be NULL
3    md_ctx = EVP_MD_CTX_create();   <-- missing check
4    if (EVP_DigestInit(md_ctx, mdtype) && ...)
```

Fig. 14: Example of null pointer dereference

Security impact. We analyze the security impact of bugs detected by GPTAid, including those in D_{comp} and APIMU4C, categorized using the CWE standard. These bugs can be classified into several types: NULL pointer dereference (CWE-476), double free (CWE-415), and improper resource shutdown or release (CWE-404). These bugs can lead to serious security implications, such as information leaks, memory corruption, crash, code execution, and so on. We use a misuse as a case study to show a common security impact in Section V-G.

G. Case study

Php-src [42] (36.6k stars on GitHub) is the source code for a popular scripting language. GPTAid found an API misuse in php-src causing a NULL pointer dereference, potentially leading to a crash. The code snippet causing misuse and the APSR generated by GPTAid are shown in Figure 14. Specifically, the APSR requires that parameter 1 of this API must not be NULL. Therefore, the caller of this API should check if parameter 1 is NULL before calling this API. In the source code of php-src, the variable `md_ctx` is obtained by calling `EVP_MD_CTX_create` (line 2), and is then passed directly to `EVP_DigestInit` (line 3) without checking whether it is NULL first, thereby violating the APSR for `EVP_DigestInit`. This results in a null pointer dereference when `EVP_MD_CTX_create` fails, leading to a system crash. However, this APSR is absent in the documentation, and the provided example of API usage is incorrect. Therefore, documentation-based approaches like Advance cannot generate this APSR, leading to the missing of this API misuse. Additionally, since `EVP_DigestInit` is only called once, detecting misuse through comparing different API usage patterns in the API calling code is not possible.

VI. Discussion

In this section, we begin by presenting our exploration of using GPT-4 for APSRs generation and using GPTAid on new APIs. We then share insights into prompt design with LLM, and lessons for preventing API misuse. Finally, we summarize GPTAid's limitations and outline future work.

A. Exploration on GPT-4

GPT-4 is a powerful LLM that outperforms GPT-3.5. We explored whether using GPT-4 directly can achieve better results in generating APSRs. We provided the same prompt used for raw APSRs generation to GPT-4 and compared the results with GPTAid (using GPT-3.5) on D_{gt}. GPT-4 achieves a recall of 0.67 and a precision of 0.21, compared to the 0.71 recall and 0.92 precision of GPTAid. These results are significantly lower than those of GPTAid, demonstrating GPTAid's effectiveness in generating APSRs. Our analysis

shows that GPT-4 is more prone to the stereotypical inference problem mentioned in Section V-C than GPT-3.5, leading to more missed APSRs.

B. Exploration on new APIs

Since LLM is trained on a large amount of data that may include API usage code, we aimed to explore whether LLM relies on this data to generate correct API calling code. Considering the training data for gpt-3.5-turbo-0613 [43] is up to September 2021, we identified 10 new APIs from eight libraries added after that date, and used GPTAid to generate their API calling code. Based on our analysis, we found that GPTAid successfully generated code for 9 out of 10 new APIs. The API that GPTAid failed to generate code for required a specific file structure as input, which is difficult to satisfy by only modifying the code. The issue was not due to the new API itself, demonstrating the effectiveness of GPTAid on API calling code generation.

C. Lessons for prompt design

Insufficient information in prompt leads to fabrication. We observed that LLM tends to generate fabricated results when there is a lack of information or clear instructions. Therefore, we recommend providing additional details when fabricated information is found in LLM's output. Furthermore, we found that using adjectives to describe expected outputs confuses LLM, leading to unexpected results. Avoiding such adjectives improves LLM comprehension of instructions.

Example in prompt might be harmful. Including examples in prompts is sometimes necessary for few-shot learning or format demonstration. However, inappropriate examples can significantly reduce LLM efficiency. For example, in the APSRs generation task, using a few-shot approach with specific APSRs examples caused the LLM to focus only on those types in the examples, missing other types. Therefore, it is crucial to ensure that examples do not limit the LLM's capabilities.

Enhance prompt design with LLM. Leveraging LLM for prompt design can be highly beneficial. Providing the LLM with the intended purpose, input/output descriptions, and asking LLM to generate the prompt can help generate high-quality and well-structured task instructions.

D. Lessons for preventing API misuse

Based on our analysis, we observe that error-prone or poorly documented documentations increase the likelihood of misuse. API developers should be mindful that users with varying experience may lack some common-sense knowledge and comprehensive security rule documentation is helpful. Additionally, careful error-checking in API calling code examples is crucial to provide accurate guidance to users.

E. Limitation and future work

For APSRs generation, GPTAid relies on dynamic analysis to determine whether an APSR is correct. However, a common limitation of dynamic analysis is that monitors may fail to capture exceptions or detect bugs that don't cause exceptions, leading to missed APSRs. The monitors used by GPTAid focus on memory issues and might miss a few APSRs, such as those related to logic errors (which do not cause exceptions) and lock-missing-unlock issues (whose exceptions cannot be captured). Some existing studies [44], [45] have extended the scope of dynamic analysis by enhancing capabilities of monitors. We plan to design more powerful monitors in the future to cover more APSRs. For bug detection, relying solely on intra-procedural analysis cause GPTAid to miss some bugs, and we plan to improve the method of detection in future work. Additionally, we plan to explore the upper limit on the number of API misuses in software to better assess the effectiveness of detection efforts and help detect more API misuses.

VII. Related Work

Recently, numerous approaches have emerged for generating APSRs and detecting API misuse, categorized as follows.

Code-analysis based approaches. Some approaches generate APSRs by analyzing the source code of library APIs. Lyu et al. [4] identifies functions related to memory management operations by analyzing the data flow of the source code and detects UAF and double free bugs. Nguyen et al. [46] use phrase-based Statistical machine translation (SMT) to translate code to the complete BE documentation. Some approaches [1], [2] summarize some heuristic rules to extract parameter rules related to exception from API source code using static code analysis techniques. Hu et al. [8] utilizes static code analysis techniques to extract API rules about return value from library source code. Unlike these approaches, GPTAid utilizes LLM to analyze library code for APSR generation, without relying on specific code patterns. Some approaches generate APSRs by analyzing client code that calls the library API.

Some approaches [10], [9] mine the correct usage patterns by identifying how most APIs are used. Wen et al. [12] mutate API usage in the client code according to predefined mutation rules and identify incorrect usage patterns by observing execution results to find API misuses. Liu et al. [11] detect bugs by checking the inconsistent security operations in a path-pair. Unlike these approaches, which are constrained by the limited APIs in the client code, resulting in the generation of restricted rules, GPTAid employs LLM to analyze the library code and generate APSRs. Some approaches derive Finite State Automata (FSA) from test case execution traces to outline the specifications [47], [48]. Unlike these methods, GPTAid analyzes API source code to formulate detailed rules, not limited to call sequence specifications but also detailing constraints on parameter values.

Text-analysis based approaches. Some approaches use NLP techniques to generate APSRs from documentation. Lv et al. [6] locate sentences with strong sentiment in documentations, and then extract API security rules by mining frequent patterns in these sentences. Ren el al. [7] uses NLP techniques and heuristic algorithms to extract information from documentations, forming a fine-grained knowledge graph of API constraints. Unlike these studies, which are limited by

documentations and can only extract a limited number of rules, GPTAid generates more rules by analyzing the library code.

VIII. Conclusion

We presented GPTAid, the first work, to the best of our knowledge, to automatically generate APSRs by analyzing API source code with LLM and detect API misuse. GPTAid utilizes an execution feedback-checking approach and code differential analysis to generate correct and concrete APSRs, and detect API misuse using APSRs. On eight popular libraries, GPTAid generated 579 APSRs, which were further investigated to enrich the documentation. Additionally, GPTAid also found 210 unknown security bugs on 47 applications integrating these libraries which can lead to system crash and denial of services. The result shows that GPTAid is capable of protecting the safety use of APIs and it enlightens the future research work on exploiting LLMs for vulnerability detection.

Acknowledgements

We would like to express our gratitude to our shepherd and reviewers for their valuable feedback, which greatly enhanced the quality of our paper. The IIE authors are supported in part by NSFC (U24A20236, 92270204), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), Youth Innovation Promotion Association CAS and by the Open Research Fund of Joint Laboratory on Cyberspace Security, China Southern Power Grid (Grant NO. GDKJXM20230642).

References

[1] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 27–37.

[2] H. Zhong, N. Meng, Z. Li, and L. Jia, "An empirical study on api parameter rules," in 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020, pp. 899–911.

[3] J. Jiang, J. Wu, X. Ling, T. Luo, S. Qu, and Y. Wu, "Appminer: Detecting api misuses via automatically mining api path patterns," in 2024 IEEE Symposium on Security and Privacy (SP). Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 43–43. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00043

[4] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu, and J. Li, "Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis," in 2022 IEEE Symposium on Security and Privacy (SP), 2022, pp. 2096–2113.

[5] M. Lin, K. Chen, and Y. Xiao, "Detecting API post-handling bugs using code and description in patches," in 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 3709–3726. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/lin

[6] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 1837–1852.

[7] X. Ren, X. Ye, Z. Xing, X. Xia, X. Xu, L. Zhu, and J. Sun, "Api-misuse detection driven by fine-grained api-constraint knowledge graph," in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020, pp. 461–472.

[8] P. Hu, R. Liang, Y. Cao, K. Chen, and R. Zhang, "AURC: Detecting errors in program code and documentation," in 32nd USENIX Security Symposium (USENIX Security 23). Anaheim, CA: USENIX Association, Aug. 2023, pp. 1415–1432. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/hu

[9] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 472–482.

[10] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "{APISan}: Sanitizing {API} usages through semantic {Cross-Checking}," in 25th USENIX Security Symposium (USENIX Security 16), 2016, pp. 363–378.

[11] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen, and Q. He, "Detecting missed security operations through differential checking of object-based similar paths," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1627–1644. [Online]. Available: https://doi.org/10.1145/3460120.3485373

[12] M. Wen, Y. Liu, R. Wu, X. Xie, S.-C. Cheung, and Z. Su, "Exposing library api misuses via mutation analysis," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 866–877.

[13] J. Li, X. Cheng, W. X. Zhao, J.-Y. Nie, and J.-R. Wen, "Halueval: A large-scale hallucination evaluation benchmark for large language models," 2023.

[14] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, P. Isabelle, E. Charniak, and D. Lin, Eds. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[15] R. Navigli, "Word sense disambiguation: A survey," vol. 41, no. 2, feb 2009. [Online]. Available: https://doi.org/10.1145/1459352.1459355

[16] M. Bevilacqua, T. Pasini, A. Raganato, and R. Navigli, "Recent trends in word sense disambiguation: A survey," in Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, Z.-H. Zhou, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2021, pp. 4330–4338, survey Track. [Online]. Available: https://doi.org/10.24963/ijcai.2021/593

[17] "openssl documentation," https://www.openssl.org/docs/manmaster/man3/, 2023.

[18] "sqlite3 documentation," https://www.sqlite.org/c3ref/funclist.html, 2023.

[19] "libpcap documentation," https://www.tcpdump.org/manpages/pcap.3pcap.html, 2020.

[20] "libxml2 documentation," https://gnome.pages.gitlab.gnome.org/libxml2/devhelp/, 2023.

[21] "libevent documentation," https://libevent.org/doc/, 2023.

[22] "libzip documentation," https://libzip.org/documentation/, 2024.

[23] "zlib documentation," https://www.zlib.net/manual.html, 2023.

[24] "libcurl documentation," https://curl.se/libcurl/, 2024.

[25] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 5–14.

[26] "Openai," https://openai.com/, 2023.

[27] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022.

[28] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[29] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[30] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[31] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," 2023.
[32] ——, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 959–971. [Online]. Available: https://doi.org/10.1145/3540250.3549101
[33] "Codeql," https://codeql.github.com/, 2023.
[34] "libxml2 documentation," https://gnome.pages.gitlab.gnome.org/libxml2/devhelp/libxml2-parser.html#xmlSAXParseMemory, 2023.
[35] "Asan," https://github.com/google/sanitizers, 2023.
[36] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 2271–2287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou
[37] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: Fuzz driver generation at scale," in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 975–985. [Online]. Available: https://doi.org/10.1145/3338906.3340456
[38] "Tree-sitter," https://github.com/tree-sitter/tree-sitter, 2023.
[39] "Valgrind," https://valgrind.org/, 2023.
[40] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu, "An empirical study on api-misuse bugs in open-source c programs," in 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), vol. 1, 2019, pp. 11–20.
[41] "Evp_digestinit_ex documentation," https://www.openssl.org/docs/man1.1.1/man3/EVP_DigestInit_ex.html, 2023.
[42] "php-src," https://github.com/php/php-src, 2024.
[43] "gpt-3-5-turbo," https://platform.openai.com/docs/models/gpt-3-5-turbo, 2023.
[44] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs," Proc. ACM Program. Lang., vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485533
[45] Q. Zhang, X. Bai, X. Li, H. Duan, Q. Li, and Z. Li, "ResolverFuzz: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing," in Proceedings of the 33rd USENIX Security Symposium, ser. USENIX Security '24, 2024.
[46] H. A. Nguyen, H. D. Phan, S. S. Khairunnesa, S. Nguyen, A. Yadavally, S. Wang, H. Rajan, and T. Nguyen, "A hybrid approach for inference between behavioral exception api documentation and implementations, and its applications," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3560434
[47] T.-D. B. Le and D. Lo, "Deep specification mining," ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 106–117. [Online]. Available: https://doi.org/10.1145/3213846.3213876
[48] H. J. Kang and D. Lo, "Adversarial specification mining," ACM Trans. Softw. Eng. Methodol., vol. 30, no. 2, jan 2021. [Online]. Available: https://doi.org/10.1145/3424307

IX. Appendix

A. Detail Information

In this section, we first present the results of our analysis on whether known API misuses trigger runtime errors detectable by monitoring tools, as shown in Table VIII. We also provide details of the libraries and applications used during evaluation in Table IX. Finally, we present the details of all the API misuses identified by GPTAid, as shown in Table X.

TABLE VIII: Study of known API misuse

Work	CWE	Impact	Catch?	#Bugs	Total
Advance	CWE-404	DoS/Information Leakage	Sanitizer	119	749
Advance	CWE-690*	crash*	runtime error	13	
Goshawk	CWE-415*	Memory Errors*	Sanitizer	45	
Goshawk	CWE-416*	Memory Errors*	Sanitizer	47	
APP-Miner	CWE-476*	crash	runtime error	89	
APP-Miner	CWE-248*	crash	runtime error	60	
APP-Miner	CWE-404*	DoS	Sanitizer	4	
APHP	CWE-911	DoS*	Sanitizer	222	
APHP	CWE-401	DoS*	Sanitizer	61	
APHP	CWE-690	crash*	runtime error	6	
APHP	CWE-404	DoS*	Sanitizer	83	
Advance	/	malfunction	/	4	47
Advance	CWE-253	authentication errors	/	3	
APP-Miner	CWE-414*	data modification	/	2	
APP-Miner	CWE-190*	resource consumption	/	1	
APP-Miner	CWE-563*	quality degradation	/	1	
APHP	CWE-235	/	/	36	

[1] / means we cannot determine this result.
[2] * indicates results derived from our analysis of CWE and vulnerabilities, while absence of * indicates that the result is explicitly stated in the paper.

TABLE IX: Libraries and Applications information

Library	Functionality	#API	Applications
libpcap	Network	71	masscan, SoftEtherVPN, nmap, john, ntopng, n2n, zmap, srs, tcpdump, freeradius-server
libxml2	XML parser	1614	openscap, php-src, aria2, collectd, postgres, vlc, ImageMagick, gpdb, ntopng, gdal
sqlite3	Database	294	netdata, php-src, leveldb, owntone-server, sqlcipher, ntopng, fluent-bit, gdal, wcdb, freeradius-server
openssl	Cryptography	5478	netdata, php-src, redis, curl, openssl, srs, SoftEtherVPN, nmap, fluent-bit, freeradius-server
libevent	Event handling	401	bitcoin-abc, evpp, owntone-server, seafile, transmission, libevent, gpdb, openvpn, kvrocks, bitcoin
libzip	File Compression	120	openrct2, hermes, ogre, monster-mash, radare2, rizin, xournalpp, idevicerestore, julius, cockatrice
zlib	Data Compression	69	openrct2, netdata, imagemagick, curl, radare2, httpd, aria2, gpdb, postgres, kvrocks
libcurl	Network Transfer	76	curl, transmission, freeradius-server, gdal, openscap, gpdb, ntopng, collectd, fluent-bit, php-src

TABLE X: List of API misuse reported by GPTAid

Library	Software	API	APSR	Impact	#Bugs
libpcap	tcpdump	pcap_compile	P_2: R_1	Dos	2
		pcap_findalldevs	P_1: R_1	Dos	1
	zmap	pcap_compile	P_2: R_1	Dos	1
	ntopng	pcap_datalink	P_1: R_3	crash	1
	n2n	pcap_compile	P_2: R_1	Dos	1
	nmap	pcap_findalldevs	P_1: R_1	crash	1
libxml2	php-src	xmlParseURIReference	P_1: R_3	crash	3
	openscap	xmlXPathEval--Expression	P_1: R_3	crash	2
sqlite3	netdata	sqlite3_open	P_2: R_2	Dos	1
	gdal	sqlite3_open	P_2: R_2	Dos	1
	fluent-bit	sqlite3_open	P_2: R_2	Dos	1
	wcdb	sqlite3_open_v2	P_2: R_2	Dos	1
	sqlcipher	sqlite3_open	P_2: R_2	Dos	1
	freeradius-server	sqlite3_open_v2	P_2: R_2	Dos	1
	owntone-server	sqlite3_open	P_2: R_2	Dos	1
openssl	nmap	EVP_DigestInit	P_1: R_3	crash	1
		SSL_CTX_set_cipher_list	P_1: R_3	crash	1
		EVP_CIPHER_CTX_set_padding	P_1: R_3	crash	2
		BN_bin2bn	P_1: R_3	crash	1
		EVP_EncryptInit_ex	P_1: R_3	crash	1
		EVP_DecryptInit_ex	P_1: R_3	crash	1
		EVP_DigestInit_ex	P_1: R_3	crash	2
	freeradius-server	EVP_DigestSignInit	P_1: R_3	crash	2
		X509_STORE_CTX_set_ex_data	P_1: R_3	crash	1
		EVP_DecryptInit_ex	P_1: R_3	crash	2
		EVP_EncryptInit_ex	P_1: R_3	crash	2
		EVP_CIPHER_CTX_set_key_length	P_1: R_3	crash	1
		EVP_DigestInit_ex	P_1: R_3	crash	3
	fluent-bit	PEM_read_bio_PrivateKey	P_1: R_3	crash	1
	php-src	EVP_DigestInit	P_1: R_3	crash	2
	openssl	SSL_ctrl	P_1: R_3	crash	3
		EVP_DigestInit_ex	P_1: R_3	crash	2
	srs	SSL_CTX_set_cipher_list	P_1: R_3	crash	1
openssl	SoftEther--VPN	EVP_DigestInit_ex	P_1: R_3	crash	1
		DH_set0_pqg	P_1: R_3	crash	1
		BN_bn2bin	P_2: R_3	crash	1
		SSL_CTX_set_verify	P_1: R_3	crash	1
		SSL_set_ex_data	P_1: R_3	crash	1
		SSL_set_fd	P_1: R_3	crash	1
		SSL_CTX_set_options	P_1: R_3	crash	7
		SSL_CTX_set_ssl_version	P_1: R_3	crash	3
	netdata	EVP_DigestInit_ex	P_1: R_3	crash	2
		SSL_CTX_get_options	P_1: R_3	crash	1
		SSL_CTX_set_options	P_1: R_3	crash	1
	redis	ERR_error_string_n	P_2: R_3	crash	1
		SSL_CTX_set_options	P_1: R_3	crash	1
libevent	kvrocks	evdns_base_resolv_conf_parse	P_1: R_3	crash	1
		evdns_base_set_option	P_1: R_3	crash	1
		evdns_base_nameserver_ip_add	P_1: R_3	crash	1
		evdns_base_nameserver_add	P_1: R_3	crash	1
	evpp	event_add	P_1: R_3	crash	2
		evhttp_connection_base_new	P_1: R_3	crash	3
		evhttp_make_request	P_1: R_3	crash	2
		evhttp_uri_get_path	P_1: R_3	crash	1
		bufferevent_setcb	P_1: R_3	crash	2
	owntone-server	event_add	P_1: R_3	crash	4
		bufferevent_free	P_1: R_3	crash	1
		bufferevent_setcb	P_1: R_3	crash	1
	seafile	event_add	P_1: R_3	crash	2
	transmi--ssion	evhttp_set_allowed_methods	P_1: R_3	crash	1
	libevent	bufferevent_setcb	P_1: R_3	crash	32
		bufferevent_getfd	P_1: R_3	crash	1
		bufferevent_pair_get_partner	P_1: R_3	crash	1
		bufferevent_enable	P_1: R_3	crash	4
		bufferevent_get_input	P_1: R_3	crash	1
		bufferevent_get_output	P_1: R_3	crash	2
		evhttp_connection_set_timeout	P_1: R_3	crash	1
		evdns_base_nameserver_ip_add	P_1: R_3	crash	17
		evdns_base_resolv_conf_parse	P_1: R_3	crash	2
		event_base_dispatch	P_1: R_3	crash	3
		evdns_base_set_option	P_1: R_3	crash	2
		event_add	P_1: R_3	crash	48
		evhttp_make_request	P_1: R_3	crash	1
		evdns_base_nameserver_add	P_1: R_3	crash	1
		bufferevent_setwatermark	P_1: R_3	crash	1

[1] APSR are abridged to improve clarity and fit within the table.
[2] P_n denotes Parameter n.
[3] R_1 indicates the rule: must be freed later.
[4] R_2 indicates the rule: must be closed later.
[5] R_3 indicates the rule: must not be NULL.

B. LLM Strategies Selection

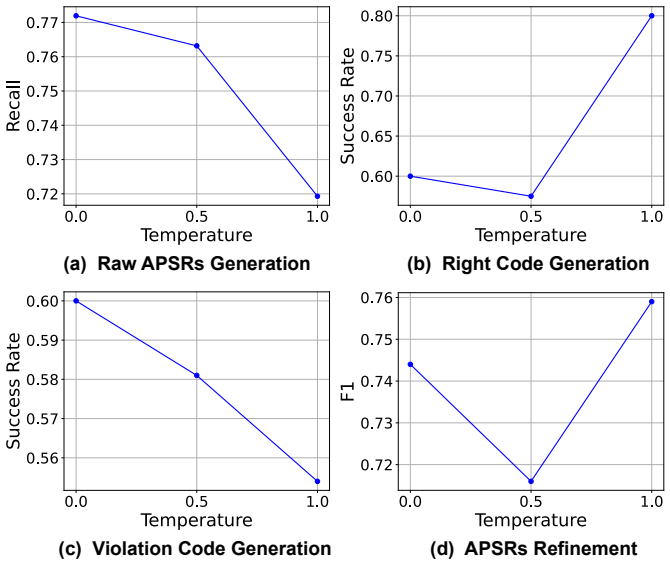

Fig. 15: Results of Temperature selection

• *Ground-Truth dataset for LLM Strategies Selection (D_{sgt}).* To select the temperature of model, we randomly selected 40 APIs from four libraries that differ from the APIs in D_{gt}. We then manually generated 134 APSRs by analyzing the API source code and documentation.

Prompt design. In this section, we assessed how different prompting methods (zero-shot, few-shot, Chain-Of-Thought) affect raw APSR generation on D_{sgt}. The recall rates were 48.20% for zero-shot, 65.80% for few-shot, and 53.50% for Chain-Of-Thought. Although few-shot had the highest recall, it mostly created APSRs types similar to those in the prompts and ignored others, which could potentially limit LLM's functionality. To foster diverse APSRs generation, we used the Chain-Of-Thought method for prompt design.

Temperature selection. In this section, we evaluated the effectiveness of GPTAid on D_{sgt} using different temperatures between 0 and 2. After several experiments, we found that setting the temperature above 1 results in meaningless, messy output. Therefore, we only conducted experiments with three temperatures: 0, 0.5, and 1. For each of the four takss using LLMs: Raw APSRs Generation, Right Code Generation, Violation Code Generation, APSRs refinement, we evaluated their effectiveness on D_{sgt}. For the Raw APSRs Generation, our goal is to maximize the number of APSRs, so we employ Raw APSR recall as an evaluation metric to identify the temperature generating the maximum number of Raw APSRs. For Right Code Generation and Violation Code Generation, we aim for LLM to generate accurate code in accordance with the specified requirements, so the success rate of code generation serves as the evaluation metric. In the APSR Refinement, our goal is to refine the APSRs to make them concrete. We focus on two key metrics: precision and recall. To assess the performance, we use the F1 score of the generated APSRs. The experimental results are shown in Figure 15. The best performance is obtained when the temperatures for the four tasks are 0,1,0,1 respectively, which is the temperature used by GPTAid.