APICAD: Augmenting API Misuse Detection through Specifications from Code and Documents

Xiaoke Wang, Lei Zhao*
Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education,
School of Cyber Science and Engineering, Wuhan University, Wuhan, China
Email: xkernel@whu.edu.cn, leizhao@whu.edu.cn

***Abstract*—Using API should follow its specifications. Otherwise, it can bring security impacts while the functionality is damaged. To detect API misuse, we need to know what its specifications are. In addition to being provided manually, current tools usually mine the majority usage in the existing codebase as specifications, or capture specifications from its relevant texts in human language. However, the former depends on the quality of the codebase itself, while the latter is limited to the irregularity of the text. In this work, we observe that the information carried by code and documents can complement each other. To mitigate the demand for a high-quality codebase and reduce the pressure to capture valid information from texts, we present APICAD to detect API misuse bugs of C/C++ by combining the specifications mined from code and documents. On the one hand, we effectively build the contexts for API invocations and mine specifications from them through a frequency-based method. On the other hand, we acquire the specifications from documents by using lightweight keyword-based and NLP-assisted techniques. Finally, the combined specifications are generated for bug detection. Experiments show that APICAD can handle diverse API usage semantics to deal with different types of API misuse bugs. With the help of APICAD, we report 153 new bugs in Curl, Httpd, OpenSSL and Linux kernel, 145 of which have been confirmed and 126 have applied our patches.**

I. INTRODUCTION

Modern software relies on various libraries by invoking Application Programming Interfaces (APIs). When using an API, users should be aware of its usage specifications. Violation of API specifications will not only affect software functionalities but also introduce security threats [1], [2]. However, due to the complexity and diversity of software development scenarios, it is challenging for users to use APIs correctly in practice [3]–[7]. As a result, API misuse bugs are widespread [3], [5] and contribute to 17.05% of all bugs [8].

To help developers correctly use APIs and help testers find and fix API misuse, many techniques for API misuse detection have been proposed. In general, detecting API misuse bugs can be seen as a process to check whether the API usage semantics violate specifications. According to the sources of constructing API specifications, existing techniques can be classified into *manual-based*, *code-based* and *text-based*.

For *manual-based* techniques, they need to manually provide API specifications or write hard-coded rules for checkers to detect defects in the target code [2], [9]–[12]. However, writing precise API specifications or specific rules relies on expert knowledge, and it is a time-consuming task even for an experienced user, making *manual-based* techniques impractical for regular users, and it is hardly scalable for the growing number of projects and their APIs.

To automatically construct API specifications, *code-based* techniques attempt to mine API specifications from code [13]–[17]. As a typical work, APISAN [16] treats deviations from the majority usage of an API as misuses since the lack of knowledge about correct API usage in existing code. Therefore, its effectiveness relies on the size and quality of the codebase it is mining [8], [18]. In other words, a codebase should contain enough API cases to reveal different types of usages, and the majority usage of different types should be correct. However, the above two conditions are not always satisfied [8]. To mitigate the codebase problem, a recent work ARBITRAR [18] leverages active learning to introduce user feedback in the detection process, but its effectiveness could still be influenced by the user's knowledge of the API.

Moreover, we can also mine *text-based* sources of specifications, including online technical forums, code comments, or standard documents. With the development of natural language processing (NLP), inferring specifications of an API from its related texts in human language seems to be a promising direction [19]–[23]. However, human language is usually diverse in writing style, and many descriptions are loosely formatted or outdated [4], [7], [24]–[26]. What is worse, many functions do not even exist in the standard documents. For example, OpenSSL [27] lists thousands of functions in *libcrypto* which are not documented [28]. In addition, it is also time-consuming to make labeled data for training models. These factors bring negative impacts to *text-based* techniques.

In summary, the automatic way is usually mining API specifications from code or texts. However, *code-based* techniques regard the majority usage as specifications, which is restricted by the codebase. By contrast, *text-based* techniques cannot ensure the integrity of constructed specifications because the diversity and loose-formatted texts limit them.

In this paper, we present APICAD, a static tool for detecting API misuse bugs of C/C++ based on code and documents. APICAD not only mitigates the demand on a large codebase with majority valid usages of *code-based* techniques but also reduces the pressure to mine information from texts of *texts-based* techniques. In brief, given a codebase to be checked and its documents, APICAD automatically executes two parallel

*Lei Zhao is the corresponding author.

jobs. In the first job, APICAD uses under-constraint symbolic execution [29], [30] to build the contexts of API invocations from the codebase and then analyzes them to extract the most-frequency usage as API specifications. In the other one, APICAD first filters the usage directive sentences in documents and then infers API specifications from documents with the help of Part-of-Speech (PoS) tagging and dependency parsing. After the above two jobs, we make a combination to acquire proper API specifications via logical disjunction and the union operation. At last, given the usage feature for each trace of the target API in a codebase, APICAD judges whether it is API misuse according to the combined API specifications with several heuristical rules to produce the final bug reports.

The key insight behind our approach is that the specification from documents can be a valid conviction as a supplement to the specification from code, and vice versa. For example, an API provided by *glibc* named *mprotect()* returns 0 if successful or -1 if not, so there are at least two valid ways to check for failure: *!= 0* or *== -1*. However, if one usage of them (e.g., *!= 0*) occurs most in a given codebase, *code-based* techniques like APISAN will treat the minor but correct cases (i.e., *== -1*) as bugs. In fact, we find that "The mprotect function returns 0 on success and -1 on failure" is briefly indicated in the documents of *mprotect()* [31]. If we leverage it to supplement the specifications mined from code, we can easily avoid the above potential false positives. On the other hand, the description of some API specifications mined from code can be absent in documents, while the information carried by code is always available, which can supplement the missing information in documents. Therefore, by combining the specifications from both code and documents, we can augment API misuse detection to be more practical.

To demonstrate the effectiveness of APICAD, we evaluate it on APIMU4C [8], which is a benchmark for API misuse detection, and four real-world programs (Curl [32], Httpd [33], OpenSSL [27] and Linux kernel [34]). Experiment results show that APICAD is efficient in building the context of API invocations in C/C++ programs and analyzing documents for detection. From the evaluation results on 2172 manually crafted API misuse bugs in APIMU4C, APICAD outperforms in detecting different types of API misuse bugs than the other three static tools. On the real-world parts of APIMU4C, the precision and recall of APICAD achieve 43.14% and 66%, respectively, increasing 30.37% and 48% compared with APISAN. In addition, we reported 153 previously unknown API misuse bugs for the chosen real-world programs, with 145 confirmed and 126 of them applied our patches.

Overall, our main contributions are as follows:

- We investigate the shortcomings of sources for automatically constructing API specifications and present APICAD, a tool for API misuse detection of C/C++ by combining the specifications from code and documents.
- We propose approaches to effectively build the context of API invocations and mine the common types of API specifications from code and documents by frequency-based mining and NLP-assisted techniques, respectively.
- We perform a comprehensive evaluation for APICAD. Results show that APICAD can capture and handle diverse API usage semantics. With the help of APICAD, we report 153 new bugs, and 145 of them have been confirmed.

Our prototype implementation is available as an open-source project at https://github.com/x2018/apicad_public.

II. Approach

Figure 1 shows the overview of APICAD, which has four modules to work. The encoded context building aims to infer the usage semantics from the context of API invocations, which can be used to mine specifications and serve as target objects for detection. To save cost and ensure effectiveness, APICAD first performs under-constrained symbolic execution to selectively explore paths for generating feasible traces to each API call site. Then the usage semantics of the traces are extracted and dumped into feature vectors to represent encoded contexts (Section II-A). Afterward, in the second module, APICAD statistically mines three types of API specifications based on the frequency of the corresponding features in the encoded contexts of an API (Section II-B).

Meanwhile, different from the above two modules which analyze code, the third module is in parallel with them for analyzing documents. Given standard documents of an API, APICAD will preprocess them and classify the sentences to different usage types through the corresponding keyword-based templates. Furthermore, the techniques in NLP, including PoS tagging and dependency parsing, are leveraged to capture the semantics carried by the classified sentences for constructing the API specifications (Section II-C). Finally, we combine the specification from code and documents by logical disjunction and the union operation. Then given a target API, APICAD retrieves the encoded contexts of it and judges whether there are bugs among them with the combined specification as well as several heuristical rules (Section II-D).

A. Encoded context building

Properly building encoded contexts with the usage semantics feature is essential for the whole work. On the one hand, the encoded context affects the quality of the specification from code because its majority usages are mined as specifications in the subsequent processing. On the other hand, to detect API misuse bugs, we need to acquire the usage from the context of API invocations on different execution paths for judgment. If the defect paths are not effectively covered, it is impossible to detect the corresponding bugs.

The invocation context for a given target call site varies according to incoming and outcoming paths. However, exploring all possible paths to a call site is infeasible due to the path explosion problem. In this work, we choose under-constrained symbolic execution [29], [30] to trace the execution paths in a limited scope to mitigate the cost.

1) Identify the analysis scope for API invocations: Before starting to execute symbolically, the analysis scope should be identified. Specifically, APICAD uses a static slicer to narrow the analysis scope by backward searching the call graph from

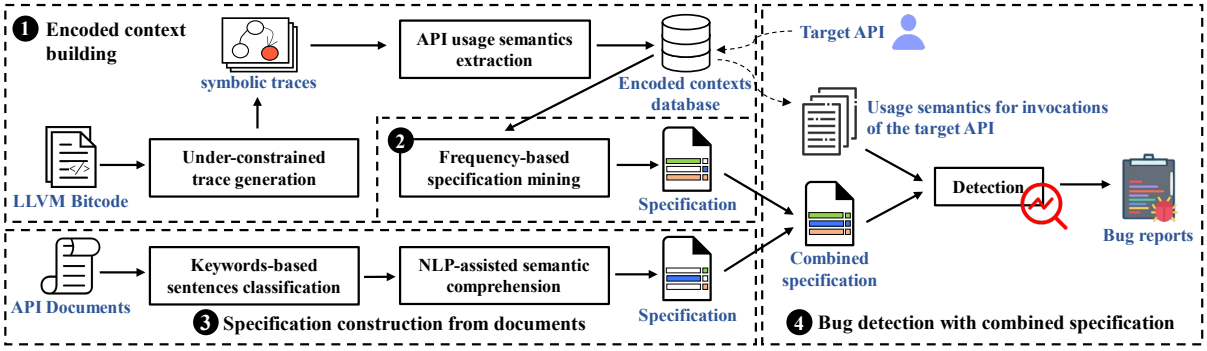

Fig. 1. Overview of APICAD. There are two parallel jobs in the left part. ❶ and ❷ are for code analysis. These two modules are essential for ❹ since we must have code to be detected. By contrast, ❸ is for document analysis and can be optional for detection as documents are not always available.

callers of the target call site. Each slice records a call chain which is from an entry to a target call, and we take *slice depth* to control the number of edges searched up. As shown in Figure 2(a), given a target call site *T* which is called by *#7* or *#8*, then slices *[#7, T]* and *[#8, T]* are generated when the *depth* is zero. While if the *depth* is one, then the edges to *#7* and *#8* are further searched to create new slices.

Because not all calls on a path are related to the usage of the target call, we further determine which calls in the scope should be analyzed in addition to the basic scope. Considering the usage of API is reflected in the propagation of its arguments and return, we make slices record *usage-related* calls whose arguments or return have relations with the target's to guide the exploration. To recognize *usage-related* calls, we perform a conservative flow-insensitive points-to analysis for the caller of the target call to lighten the analysis burden. In the analysis, whether a variable is local or global, we see the allocation of it as a memory location and track the propagation of it by unwrapping the operand of instructions such as *Load*, *Store* and *GEP*. After gathering all possible propagation of the memory locations, the *usage-related* calls are filtered by judging whether their arguments or return can be propagated from the memory location that propagates to the target.

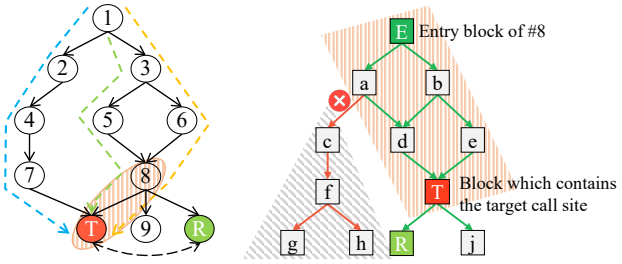

Fig. 2. API invocation context generation.

2) Build the context of API invocations: The primary setting of under-constraint symbolic execution is inherited from ARBITRAR [18] that uses a hash-based memory model, unrolls each loop once, and does not repeatedly explore recursive function. The format of a trace generated by it is shown in Table I, which is a sequence of instructions on a program path with symbolic or concrete values. However, since it only executes the loop once and does not execute some internal calls, many values can influence the path condition we cannot calculate. To avoid missing possible paths, we replace the result of the binary operation with a new symbol after entering a loop because these binary operations can influence the branches inside the loop. Besides, if a call we do not step into takes memory addresses as arguments, we also replace the value in the memory with a new symbol since it is unclear how the call operates on the value.

TABLE I
THE FORMAT OF A SYMBOLIC TRACE.

Function	f		
Integer	α		
Symbolic variable	β		
Symbolic type	τ	::=	[arg | local | global | symbol]
Symbolic expression	e	::=	$[\alpha \mid \beta_\tau]$
Boolean	*bool*	::=	[true | false]
Binary operation	*binop*	::=	[+ | − | × | ÷ | %]
Comparison condition	*cond*	::=	[= | ≠ | ≤ | < | ≥ | >]
i-th event t_i on the trace	::=	*Call*	$(\mathrm{i}, \mathrm{e}_{\mathrm{result}}, \mathrm{f}, \overline{\mathrm{e}}_{\mathrm{args}})$
	|	*Assume*	$(\mathrm{i}, \mathrm{cond}, \mathrm{e}_1, \mathrm{e}_2, \mathrm{bool})$
	|	*Store*	$(\mathrm{i}, \mathrm{e}_{\mathrm{location}}, \mathrm{e}_{\mathrm{value}})$
	|	*Load*	$(\mathrm{i}, \mathrm{e}_{\mathrm{result}}, \mathrm{e}_{\mathrm{location}})$
	|	*GEP*	$(\mathrm{i}, \mathrm{e}_{\mathrm{result}}, \mathrm{e}_{\mathrm{location}}, \overline{\mathrm{e}}_{\mathrm{indices}})$
	|	*Binary*	$(\mathrm{i}, \mathrm{e}_{\mathrm{result}}, \mathrm{binop}, \mathrm{e}_1, \mathrm{e}_2)$
	|	*Alloca*	(i, e)
	|	*Ret*	(i, e)
Target event t_k	≡	*Call*	$(\mathrm{k}, \hat{\mathrm{e}}_{\mathrm{result}}, \hat{\mathrm{f}}, \overline{\hat{\mathrm{e}}}_{\mathrm{args}})$
Symbolic trace	ρ	::=	$\{t_1, ..., t_k, ..., t_n\}$

Moreover, as shown in Figure 2(b), it is useless to explore the edge a→c as the following blocks cannot reach the target call site. Though we do not know which paths are reachable when executing forward, we know that the entry block can be reached no matter which path we take back from the target. Therefore, to save unnecessary time consumption, APICAD firstly takes a simple backward searching from the target based on the CFG to collect reachable paths and further uses them to guide the execution.

After establishing the guided paths, APICAD starts the under-constraint symbolic execution with both of them and their corresponding slices. Specifically, it executes from the entry point with fresh symbols as arguments and explores by following the guided block paths to the target call site, while the exploration directions are not restricted anymore after the target call site. During the path exploration, APICAD attempts to step into a call if the slice records the call as *usage-related*, such as *R* in Figure 2(b), and the call's implementation is available in the analyzed bitcode. Otherwise, APICAD directly

TABLE II
USAGE FEATURES EXTRACTED BY APICAD WITH THEIR DEFINITIONS AND DESCRIPTIONS

Type	Feature	Definition	Description
Return	$checked$	$\exists\, i > k : Assume(i, cond_i, e_1, e_2, bool_i) \wedge (\hat{e}_{ret} = e_1 \vee \hat{e}_{ret} = e_2) \wedge \forall\, j < i : t_j \notin \{Store(j, \hat{e}_{ret}, \_),\ Load(j, \_, \hat{e}_{ret})\}$	Return value is checked before it is dereferenced.
	chk_cond	$\neg checked \Rightarrow None, checked \wedge bool_i \Rightarrow cond_i, checked \wedge \neg bool_i \Rightarrow \neg cond_i$	Check condition for checking return value.
	cmp_value	$\neg checked \Rightarrow None, checked \wedge \hat{e}_{ret} = e_2 \Rightarrow e_1, checked \wedge \hat{e}_{ret} = e_1 \Rightarrow e_2$	Comparison value for checking return value.
	$used\_in\_call$	$\exists\, i > k : Call(i, \_, \_, \overline{e}_{args}) \wedge \hat{e}_{ret} \in \overline{e}_{args}$	Return value is used as an argument of another call.
	$used\_in\_bin$	$\exists\, i > k : Binary(i, \_, \_, e_1, e_2) \wedge (\hat{e}_{ret} = e_1 \vee \hat{e}_{ret} = e_2)$	Return value is used in a binary operation.
	$stored\_not\_local$	$\exists\, i > k : Store(i, e_{loc}, \hat{e}_{ret}) \wedge (e_{loc} = \beta_{arg} \vee e_{loc} = \beta_{global})$	Return value is stored to an address outside the caller.
	$returned$	$\exists\, Ret(n, \hat{e}_{ret})$	Return value is finally returned.
	$derefed$	$\exists\, i > j,\ j > k : Store(i, \hat{e}_{ret}, \_) \vee Load(i, \_, \hat{e}_{ret}) \vee (GEP(j, e_j, \hat{e}_{ret}, \_) \wedge (Store(i, e_j, \_) \vee Load(i, \_, e_j)))$	Return value is dereferenced to read/write.
	$indir\_returned$	$\exists\, Ret(n, e\_parent) \wedge Store(\_, e\_child, \hat{e}_{ret}) \wedge is\_child\_item(e\_parent, e\_child)$ $\mid is\_child\_item(e_1, e_2) ::= \exists\, GEP(\_, e_{result}, e_{loc}, \_) : e_{result} = e_2 \wedge (e_{loc} = e_1 \vee is\_child\_item(e_1, e_{loc}))$	Return value is stored into another address and returned.
Argument	$is\_constant_x$	$\hat{e}_{arg}[x] = \alpha$	Argument x is a constant.
	$post.returned_x$	$\exists\, Ret(n, \hat{e}_{arg}[x])$	Argument x is finally returned.
	on_stack_x	$\exists\, Alloca(\_, e) : \hat{e}_{arg}[x] = e \vee (\exists\, i < k : GEP(i, e_i, e_{loc}, \_) \wedge e_{loc} = e \wedge e_i = \hat{e}_{arg}[x])$	Argument x is a memory address on stack.
	$pre.checked_x$	$\exists\, i < k : Assume(i, \_, e_1, e_2, \_) \wedge (\hat{e}_{arg}[x] = e_1 \vee \hat{e}_{arg}[x] = e_2)$	Argument x is checked before the call site.
	$post.checked_x$	$\exists\, i > k : Assume(i, \_, e_1, e_2, \_) \wedge (\hat{e}_{arg}[x] = e_1 \vee \hat{e}_{arg}[x] = e_2) \wedge \forall\, j < i : t_j \notin \{Store(j, \hat{e}_{arg}[x], \_),\ Load(j, \_, \hat{e}_{arg}[x])\}$	Argument x is checked after the call site.
Causality	$pre.calls$	$\overline{f}_{pre} \mid \forall\, f \in \overline{f}_{pre}, \exists\, i < k : Call(i, e_i, f, \overline{e}_{args}) \wedge (f \in$ *usage-related-calls* $\vee\ (e_i \cup \overline{e}_{args}) \cap \overline{\hat{e}}_{args} \neq \emptyset)$	Causal calls which occur before the target call site.
	$post.calls$	$\overline{f}_{post} \mid \forall\, f \in \overline{f}_{post}, \exists\, i > k : Call(i, e_i, f, \overline{e}_{args}) \wedge (f \in$ *usage-related-calls* $\vee\ (\hat{e}_{ret} \cup \overline{\hat{e}}_{args}) \cap \overline{e}_{args} \neq \emptyset)$	Causal calls which occur after the target call site.

replaces the call's return value with a fresh symbol. Besides, the target call itself is also ignored to handle since we are not interested in knowing API internals. When reaching the return point of the analysis scope, a symbolic trace is generated if the path constraints are satisfiable.

3) Extracting usage semantics: According to the previous analysis for API misuse bugs [8], [16], [18], APICAD encodes each trace ρ of an API into a feature vector $encoded(\rho)$ with three types to represent various usage semantics. In general, these features are extracted by evaluating the return value and arguments of the target call on the events on a trace. Table II shows the definitions and descriptions of these features, and a brief explanation for them is as follows:

(a) Return value. The return value in C/C++ is usually used as an execution status code or a pointer to memory. Checking the return value is important to avoid security defects such as NULL dereference. Therefore, we are concerned about how the return of a call is checked in addition to the suspicious operations on the return value. Note that we argue that the return checking should be carried out before it is dereferenced so that we make $checked$ valid only when it is performed before *Load* and *Store*.

(b) Argument. Before calling an API, we may need to verify whether the argument is sensible, such as that some pointer-type arguments cannot be NULL. Therefore, we record how arguments are checked before the call. Besides, since some functions may use arguments as execution status, we are also concerned about whether they are checked after the call.

(c) Causality. There can be a causal relationship between different APIs. For instance, an allocated memory should be freed. To excavate the potential causal relationships, we still use the arguments and returns as the metric and also leverage the *usage-related* calls recorded by the corresponding slice to filter them from the trace.

In fact, several features are also defined by ARBITRAR via Datalog rules. The main differences between us are: (a) We remove many definitions, including the features used to represent control flows and for each separate causal call, because they are weakly related to usage. (b) We use chk_cond and cmp_value to handle complex assumptions rather that just being sensitive to $assumed\_zero$. (c) We optimize definitions such as $derefed$ and $checked$ to make them more precise. The change for $checked$ is explained in the above description. As for $derefed$, ARBITRAR evaluates *Load*, *Store* and *GEP* to infer it, while we do not directly see $GEP(\_, \_, e_{location}, \_)$ as a dereference to $e_{location}$ since *GEP* just calculates the memory address rather accessing the memory.

B. Frequency-based specification mining

To automatically mine specifications from code, we still follow the assumption that the majority usage is correct. Though the quality of the codebase limits this assumption, it is indeed a simple and effective practice without additional knowledge about specific APIs [16].

TABLE III
MINING SPECIFICATIONS FROM CODE. ROWS 1-5 ARE AUXILIARY FOR SHOWING THE PROCESS. ROWS 6-11 INDICATE MINED SPECIFICATIONS.

$API\ k's\ contexts$	$::= \{encoded(\rho_1), ..., encoded(\rho_n)\}$
$num(feature, val)$	$::= \sum_{i=1}^{n}(encoded(\rho_i).feature = val \vee val \in encoded(\rho_i).feature)$
$pre\_hint(call)$	$::= \exists\, hint\_str \in ['alloc', 'open', ...] : hint\_str\ in\ call's\ name$
$post\_hint(call)$	$::= \exists\, hint\_str \in ['free', 'close', ...] : hint\_str\ in\ call's\ name$
$hints(post)$	$::= post\_hint(post) \Rightarrow \{pre\_hint(k) \Rightarrow 0.3, else\ 0.1\}, else\ 0$
$ret.need\_check$	$::= num(ret.checked, true)/n > \lambda$
$ret.chkvals$	$::= \overline{v} \mid \forall val \in \overline{v} : num(ret.cmp\_value, val)/n > \lambda$
$arg.pre.need\_check_x$	$::= num(arg.pre.checked_x, true)/n > \lambda$
$arg.post.need\_check_x$	$::= num(arg.post.checked_x, true)/n > \lambda$
$causal.pre\_calls$	$::= \overline{f}_{pre} \mid \forall f \in \overline{f}_{pre}$: $num(casual.pre.calls, f)/n > \lambda$
$causal.post\_calls$	$::= \overline{f}_{post} \mid \forall f \in \overline{f}_{post}$: $num(casual.post.calls, f)/n + hints(f) > \lambda$

In general, there are six mined specifications, as demonstrated in Table III, which are also carried out by three types.

Specifically, we first cluster some sub-features based on all the encoded contexts of an API k. Then the frequency of a specific feature value is calculated by the number of its occurrences divided by the total number of traces. The feature value will be regarded as a valid specification if the frequency is higher than a threshold λ. Note that not all features are clustered to generate a specification. Some features, including *derefed* and *returned*, are only leveraged to provide contextual evidence for detection. Especially for the post-causal functions, we further evaluate them according to their checking conditions. If a causal call mostly occurs under a specific assumed value (obtained through chk_cond with cmp_value), we assume that it only needs to be called under this value. This setting takes into account a fact that an API does not need post causal calls if it fails (return a specific value). Besides, inspired by APISAN [16], APICAD takes considerations on some deterministic hints for causal relationships. For example, if a sub-string *free* or *clear* is contained in the name of a post-causal function, which means that it is a likely memory release API, so APICAD prefers to include it in the causal specification by increasing its frequency score.

$$sigmoid(x) = \frac{1}{1+e^{-x}}, x = \frac{n-\theta}{\theta/5} \tag{1}$$

$$\lambda = 0.5 + 0.3 * sigmoid(x) \tag{2}$$

For the threshold λ, we argue that it should be tolerant for the small samples and strict with the complex samples of an API. Based on our observation, the small samples of an API are more challenging to form centralized clusters statistically. Besides, it will not bring a lot of false reports even if we mine a wrong specification. By contrast, the complex samples are enough to reveal valid specifications. A higher threshold can avoid invalid specifications mined from some invocation side effects. Therefore, APICAD uses a modified sigmoid function as λ. As shown in Equation 2, the variable n denotes the number of traces for the API, and θ is a hyperparameter to control how large a range we adjust. The growth of threshold in the range of $n = [0, 2 * \theta]$ is the same as the shape of sigmoid function (Equation 1) in the range of $x = [-5, 5]$. We set the upper limit to 0.8, which is the default threshold of APISAN. Then when n is zero, λ is around 0.5, while when n is larger than $2 * \theta$, λ is approaching 0.8.

C. Specification construction from documents

In this section, we describe how APICAD constructs specifications from API documents. The main idea of the approach is the observation that API documents have the basic structure and grammatical characteristics corresponding to different usage types. To construct specifications automatically, sensitive keywords and NLP-assisted techniques can handle such structure and grammatical characteristics, respectively.

The first step is to preprocess the documents. Particularly, we collect the standard documents from websites and split the information of each API existing in them, removing usual stopped words and expanding abbreviations, etc. The information of each API is thereby dumped into several parts according to the original tag, including NAME, SYNOPSIS, DESCRIPTION, and RETURN VALUE. The SYNOPSIS is in the form of pseudo code to define the method so that APICAD analyzes it to extract the types/names of the API and its parameters. Then the descriptions of the method/parameters are divided respectively by their names.

In the second step, the usage directive sentences in respective descriptions of the method itself or each of its parameters are further filtered. This process leverages several sensitive keyword patterns to identify the sentences into corresponding categories. Though some sentences with complex patterns may be ignored, we can at least ensure that the obtained information is as precise as possible under this lightweight analysis. In addition, it does not require considerable prior knowledge, such as sufficient labeled samples, to train a model for classification so that it is more practical to implement. In brief, like the types of specifications mined from code, we still focus on the usage of return checking, parameters checking and causality: (a) To identify the sentences related to the return checking, we take all the sentences in RETURN VALUE, and use "return" to filter the sentences in DESCRIPTION. (b) The parameter pre-checking directive sentences are retrieved by the keyword templates, including "must [not] be", "should [not] be", because we find that common descriptions about parameter pre-checking satisfy the pattern, such as "must not be NULL" or "should be larger than 0". As for the parameter post-checking, we are concerned about the words about status such as "success", "fail", and "error" to filter the sentences like "A pointer to the result in case of success or NULL on error is stored in para_foo". (c) A sentence is recognized as causality related if it contains the other method names except for the analyzed method itself and it has sensitive words such as "allocate", "release", and "free".

With the usage directive sentences, we are in a position to construct usage specifications with the help of PoS tagging and dependency parsing. According to the PoS tag and dependency relationships of each word in a sentence, APICAD firstly identifies its subject, the actions of the subject and the objects of the subject, and captures their various modifiers. Then APICAD infers the semantics according to the directive type of a sentence: (a) For the return checking, the objects are extracted to be the possible values that can be returned, and the modifiers of the objects are processed to obtain the conditions for returning the respective value. If it is a negative sentence, then the condition will also be reversed by APICAD. (b) The parameter pre-checking is considered required if the parameter is the nominal subject of a value. Processing the parameter post-checking is similar to the processing of return checking, but we only care whether the parameter needs to be checked, which is determined by judging whether the parameter is used to carry the return status. (c) For the causality, if it is a passive sentence such as "must be released by foo", the oblique nominal (foo) of the passive action (released) will be recognized as a causal function of the API. Otherwise, we attempt to take the objects as possible causal functions and

identify the causal relationship through adverbial clause modifiers. Besides, we also attempt to find the explicit order words, including "before" and "after" for correcting the judgment of pre- or post-conditions.

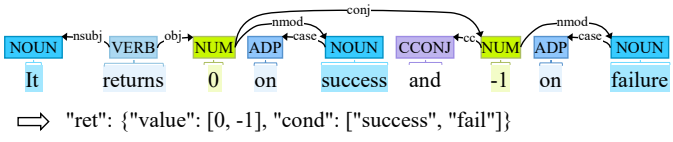

Fig. 3. A directive sentence about "return" of *mprotect()*.

For instance, Figure 3 illustrates a sentence with the PoS tagging and dependency parsing result. This sentence is identified as a directive sentence about "return" since it contains the keyword "return". In this sentence, the nominal subject (nsubj) is "It", which refers to *mprotect()*. The action in the sentence is "returns" with the "VERB" tag, and its objects are "0" and "-1" ("-1" is the conjunction of "0"). Thus, we mark the potential return values as "0" and "-1". Then "success" and "failure" can be further captured through nominal modifier (nmod) as the conditions underlying "0" and "-1", respectively. The functionality of these conditions is similar to the assumed value in the causality specification from code. If we can know what a return value is for failure, then the post causal functions will not be considered when the return value is assumed as the failure. Note that the conditions can also be captured from the other dependency relationships, such as adverbial clause modifiers and oblique nominal, and we unify the conditions as "success" and "fail" according to their keywords, such as success, successful, and error, failure. Besides, we also rewrite many common NOUNs to NUM, such as NULL→0.

After the previous steps, three types of API specifications are constructed: (a) For the return value, the specification is in the form of $\{$"$value$" : [], "$cond$" : []$\}$ and the index of the $cond$ is corresponding to the same index of $value$. (b) For the arguments, the specification is in the form of $\{$"$pre.check$" : $bool$, "$post.check$" : $bool\}$. (c) As for the causality, the specification is $\{$"pre" : [], "$post$" : []$\}$.

D. Bug detection with combined specification

As we demonstrated before, we define the same three types for the specifications from code and documents. We cannot ensure the specifications from the code or documents are sound or complete due to the natural limitations of the related techniques, so we choose the logical disjunction to combine the specifications of boolean type and use the union operation to combine the set type to give full play to the analysis fruits of both code and documents. For specifications of boolean type, logic disjunction will not let go of the defects we could have caught. For set type, the union operation makes the results obtained from the analysis of both code and documents preserved. For example, if $code.ret.need\_check = false$ but $doc.ret.value \neq \emptyset$, then we think the return value of the API is needed to be checked, while the feasible values to be checked should be $code.ret.chkvals \cup doc.ret.value$.

With the combined specification, APICAD retrieves every encoded context extracted from all the traces of a given API for detection. The primary detection logic is judging whether a specific feature violates the corresponding specifications. Specifically, for the return checking, APICAD first judges whether the check is needed. If so and it is checked, APICAD then further judges whether the check value is reasonable. Judging the argument checking is straight since there is just one boolean specification. Note that we focus on the argument that can be a memory address ($is\_constant = false$) for argument post-checking. Otherwise, the returned argument is unnecessary to be checked since the call cannot manipulate it at all. As for the causality, APICAD checks whether the functions in the specification exist in the current context. In particular, a post-causal violation will be ignored if the assumed return value of the encoded context is not recorded in the post-causality specification from code or is known as a failure value by the specification from documents.

Except for the features for comparing with the specifications, APICAD leverages the other features to expose the defects that can be revealed in a single context as well as to suppress false positives via the following rules: (a) The feature is_alloca is to alert obvious cases with common knowledge. For instance, if the API name contains *free* and is_alloca is true, APICAD will report it as it may wrongly deallocate an address on the stack. (b) If there are causal calls with the same of the target's name, we also check whether the name contains the common memory release string such as "free". If so, we will report it as a potential double-free defect. Besides, for the calls with one argument, we detect the lack of return checking if it does not have post-causal calls with the same target's name to avoid redundant judgments. (c) Due to the "short-circuiting" in C/C++, a return without checking may not be a real bug. However, we cannot perceive it based on one trace. Therefore, we tolerate the return without checking if there is a natural trace for the same call site. Besides, we do not know whether the API is successful or not if it is really under "short-circuiting", so we leverage the features including $derefed$, $used\_in\_bin$, and $used\_in\_call$ to reveal the status of the return value. If the return is not used at all, we suppose this call is in failure and ignore checking its subsequent casual functions. (d) Moreover, if the argument/return is returned or stored globally, we prevent reporting the bug since this means that APICAD lacks the context out of the current scope. Similarly, if an argument is returned, the post-checking bug of the argument is also prevented from reporting.

At last, APICAD outputs the location of all reported bugs with violation information and corresponding features. We believe these outputs are helpful for users to understand the original code and distinguish between real and false reports.

III. Evaluation

We implement a prototype of APICAD in about 4K Lines of Code (LoC). The encoded context building module is implemented using Rust with about 1.5K LoC, and it is developed based on the symbolic execution engine provided

by ARBITRAR [18]. Moreover, we use python to enable the other three modules. The PoS tagging and dependency parsing are performed with the pre-trained model of HanLP [35].

To evaluate APICAD, we conduct comprehensive experiments for our prototype with the following four objectives.

- **Functionality.** Can APICAD capture and handle different API usage semantics for detection?
- **Effectiveness.** How effective is APICAD in finding API misuse bugs on real-world programs?
- **Impact.** Can APICAD find previously unknown API misuse bugs in real-world programs?
- **Performance.** How about the performance of code and document analysis implemented by APICAD?

Apart from the above objectives, we also discuss the advantages of APICAD in the form of several case studies at the end of this section. All the experiments are running on Ubuntu 20.04 LTS with Clang 12.0.0 installed, and the machine has an 8-core CPU (Intel i7-11700, 2.50GHz) and 48 GB RAM. Since the usage of most APIs exists in the caller function [16], we set the *slice depth* to be zero by default, which is the same as ARBITRAR. Moreover, considering there are usually 500-800 traces for a fairly used API, we take $\theta = 500$ as a default evaluation setting to adjust the threshold λ.

A. Dataset

1) Bug-benchmark: We use APIMU4C [8] as our evaluation benchmark, which is an open API misuse bug benchmark. This benchmark has two parts: (a) Single-File-Case. This part contains 2172 manually crafted API misuse bugs modified from Juliet Test Suite [36] and Toyota ITC [37]. It covers three common API misuse types: IPU (improper parameter using), IEH (improper error handling), and ICC (improper causal function calling). Therefore, we choose this part to evaluate the basic functionality of APICAD. (b) Project-Case. It contains 100 bugs from the old version of OpenSSL, Curl and Httpd. Since this part provides a ground truth in its range, we take the results on this part to estimate the precision and recall of APICAD on real-world programs.

2) Real-world programs: We choose 4 well-known open source projects including Curl (master c40914db) [32], Httpd 2.4.51 [33], OpenSSL (master 0299094c) [27] and Linux kernel 5.15 [34]. Their release dates are all around November 2021. We use them to evaluate the efficiency of code analysis and the impact on finding new real-world bugs.

3) Standard documents: We obtain the standard documents of GNU libc, Linux kernel, and OpenSSL from websites because the APIs described in these documents are widely used in the chosen real-world programs. Specifically, Curl and Httpd can all be compiled with OpenSSL enabled.

B. Functionality

The Single-File-Case benchmark contains 510, 612, and 1050 bugs of IPU, IEH, and ICC, respectively, which correspond to the three types of our defined specification. It covers different types of API misuse, but all its bugs are only specific to several sensitive APIs. To be unbiased, we do not feed any particular document when evaluating this part. As a result, this part actually shows the ability of APICAD to detect manually crafted bugs based on the specification from code alone, but it is enough to reflect that APICAD supports handling different usage semantics for detecting API misuse bugs.

Table IV lists the results with precision and recall as metrics, and the data of the other three static tools are directly borrowed from the original results of APIMU4C. In general, APICAD finds more bugs on each of the three types than other tools and performs best in terms of overall precision and recall, which shows that APICAD is good at extracting different types of usage semantics for detection. APISAN [16] performs not well and even fails to detect all bugs of IPH, because it cannot support many implicit semantics [8], [18]. By contrast, CPPCHECK [11] and CLANG-SA [12] are more powerful in code analysis and report bugs conservatively, so they result in a relatively high precision. However, in addition to semantic analysis ability, they can also be limited by their individual rules/checkers written to encode API specifications [8].

```
61 char * data;
62 data = NULL; /* Initialize data */
63 /* POTENTIAL FLAW: Allocate memory without checking */
64 data = (char *)malloc(20*sizeof(char));
65 switch(6)
66 {
67 case 6:
68    /* FLAW: Initialize memory buffer without checking */
69    strcpy(data, "Initialize");
```

Fig. 4. A code snippet for explaining duplicate TP.

Note that there are 119 TPs reported as duplicates. Typically, a duplicate TP means the report is right, but the root cause of it is the same as another TP in a different location. For example, as shown in Figure 4, there are two potential flaw locations reported by APICAD, both caused by that *data* is not checked, so we regard one of them as a duplicate. We reserve these duplicate TPs because they can give users a more comprehensive perspective on a buggy code to find the root cause better. To ensure fairness, we only include the 119 TPs in the precision calculation, not recall.

C. Effectiveness

The Project-Case benchmark provides a ground truth in its range for real-world programs. Because many bugs are confirmed in later versions, we excluded them from the statistic to stay within the scope of this benchmark. Besides, to reflect the role of adjusting thresholds, there are two additional control groups with different conditions: (a) The threshold λ is set to 0.5. (b) The threshold λ is set to 0.8. Meanwhile, we list the results of disabling document analysis in parentheses to show the role of the specifications from documents.

As shown in Table V, APICAD with default setting performs better than APISAN in terms of precision and recall, and we explain why APICAD is better in Section III-D. After disabling document analysis, APICAD can no longer detect 19 bugs, and the precision of APICAD also decreases by 6.71%, which indicates that the combined specification is beneficial to

TABLE IV
EVALUATION RESULTS ON THE SINGLE-FILE-CASE BENCHMARK OF APIMU4C.

Type	Num	APISAN				CPPCHECK (ver 1.83)				CLANG-SA (ver 6.0.0)				APICAD			
		Report	TP	Precision	Recall	Report	TP	Precision	Recall	Report	TP	Precision	Recall	Report	TP	Precision	Recall
IPU	510	0	0	0	0	145	127	87.59%	24.90%	127	105	82.68%	20.59%	272	272	100.00%	52.94%
IEH	612	446	173	38.79%	28.27%	298	270	90.60%	44.12%	0	0	0	0	448	448	100.00%	73.20%
ICC	1050	447	435	97.32%	41.43%	373	337	90.35%	32.10%	746	565	75.74%	53.81%	829	757	91.32%	60.95%
Total	2172	893	608	68.09%	27.99%	816	734	89.95%	33.79%	873	670	76.75%	30.85%	**1549**	**1477↑**	**95.35%↑**	**62.52%↑**

TABLE V
EVALUATION RESULTS ON THE PROJECT-CASE BENCHMARK OF APIMU4C. THE RESULTS WITHOUT FEEDING DOCUMENTS ARE IN PARENTHESES.

Target	Num	APISAN				APICAD ($\lambda = 0.5$)				APICAD				APICAD ($\lambda = 0.8$)			
		Report	TP	Precision	Recall	Report	TP	Precision	Recall	Report	TP	Precision	Recall	Report	TP	Precision	Recall
OpenSSL	50	108	13	12.04%	26%	167 (179)	48 (47)	28.74% (26.26%)	96% (94%)	117 (98)	44 (31)	37.61% (31.63%)	88% (62%)	93 (56)	43 (23)	46.24% (41.07%)	86% (46%)
Curl	30	10	1	10%	3.33%	30 (28)	11 (8)	36.67% (28.57%)	36.67% (26.67%)	18 (16)	10 (7)	55.56% (43.75%)	33.33% (23.33%)	17 (14)	9 (6)	52.94% (42.86%)	30% (20%)
Httpd	20	23	4	17.39%	20%	42 (39)	13 (10)	30.95% (25.64%)	65% (50%)	18 (15)	12 (9)	66.67% (60%)	60% (45%)	17 (9)	12 (5)	70.59% (55.56%)	60% (25%)
Total	100	141	18	12.77%	18%	239 (246)	72 (65)	30.13% (26.42%)	72% (65%)	**153 (129)**	**66 (47)**	**43.14% (36.43%)**	**66% (47%)**	127 (79)	64 (34)	50.39% (43.04%)	64% (34%)

improve accuracy. Moreover, the result of detection only with specifications from documents is that there are 56 reports in total and 36 bugs in the benchmark are detected. Therefore, there are 30 bugs that can be only detected by code-based specifications. Apart from mitigating false negatives, the combined specification also suppresses many false positives because it can better avoid reporting valid cases, such as many failed invocations that do not require subsequent operations. When λ is 0.5 and after enabling document analysis, the total number of reports is even reduced with more true bugs reported.

Meanwhile, we can see that the higher the threshold λ, the higher the precision and the more significant the role of document analysis. Notably, when λ is 0.8, the precision is the highest, and 30 bugs can only be detected by the specification obtained from documents. That is, the stricter the specifications we get from code, the more precise the specifications are, but the more likely many valid specifications from code with relatively low frequency will be lost. With the sigmoid threshold, we make the capability of detection with code-based specifications reach a good trade-off between precision and recall. Because macro functions are expanded into ordinary instructions in the bitcode, and callback functions are not precisely identified under the current analysis, now APICAD is not sensitive to 17 bugs in this benchmark, such as the macro va_start() in *httpd*, and the callback functions for memory management in *libcurl*. Besides, there are 17 additional bugs left out by APICAD and 6 of these bugs can be detected when the threshold λ is 0.5. In Section IV, we will discuss why APICAD brings false positives and negatives.

Note that ARBITRAR also has its evaluation on this benchmark. However, it only selects part of the bugs (47 of 100) and does not make its choice available. Because there are multiple subjective factors from users to influence ARBITRAR's effectiveness, we did not make a direct comparison with it. In fact, the features extracted by APICAD can be compatible with the active learning detection module of ARBITRAR in theory. Without considering the subjective factors, some analysis strategies of APICAD are also beneficial to ARBITRAR. For example, APICAD can cover the feasible path as we discussed in Section III-F1 but ARBITRAR fails, because it unrolls the loop once but ignores to handle variables related to the loop which can influence the path constraint.

D. Impact

After getting raw bug reports with usage feature, misuse type and location from APICAD, we manually validate true reports and provide corresponding materials to maintainers. In total, we report 153 new API misuse bugs for chosen programs with the help of APICAD. Table VI lists the bugs we report. Note that many detected bugs are not listed because they had been patched when we found them. Until now, the maintainers confirmed 145 of our reported bugs, and our patches have been applied to 126 of these bugs. In particular, the reported bugs of Httpd are patched by other developers who respond to our reports in Httpd's Bugzilla. Based on our analysis, all of these bugs have a non-ignorable impact, and some of them have security implications that can cause memory crashes or memory leaks. In addition, the lack of checking for return value is the most common type in the reported bugs.

TABLE VI
IMPACT ON FINDING NEW REAL-WORLD BUGS. THE NUMBER OF PATCHED REFERS TO HOW MANY OF OUR PROVIDED PATCHES ARE APPLIED.

Programs	Reported	Confirmed	Patched	Missed by APISAN
Curl	15	15	15	14
Httpd	9	9	0	5
OpenSSL	61	61	61	45
Linux kernel	68	60	50	-
Total	153	145	126	64

To further show the detection capability of APICAD, we manually validated the results of APISAN in the same manner as the validation for APICAD based on the confirmed bugs. APISAN builds on Clang 3.6, but compiling Linux 5.15 needs the version of Clang $\geq$ 10.0.1. Therefore, we only tested APISAN on Curl, Httpd, and OpenSSL. As listed in Table VI, APISAN fails to reveal 64 of 85 confirmed bugs. There are two main aspects to explain this phenomenon. First, the context analysis of APISAN has limitations in handling complex semantics of different usage as discussed in previous works [8], [18] and APICAD mitigates them. Moreover, many real-world API misuse bugs cannot be detected by APISAN because of the codebase problem. On the one hand, APISAN

directly sets the threshold to 0.8, which can lose many valid specifications with relatively low frequency, while APICAD adjusts the threshold adaptively by considering the size of a codebase. On the other hand, many bugs can never be detected by the specification mined from code based on frequency but can be filtered by the specification constructed from documents. By contrast, APICAD gives play to the role of two sources instead of taking a single source.

E. Performance

1) Code analysis: To demonstrate APICAD is efficient and scalable to analyze regular real-world programs, we make APICAD generate encoded contexts for the whole project of Curl, Httpd, and OpenSSL. This process is a one-time task in the workflow of APICAD. All the raw traces are also stored, which can be leveraged in the follow-up analysis. For example, we can recover the path corresponding to a raw trace in source code and visualize it to help users validate our reports.

TABLE VII
ENCODED CONTEXT BUILDING FOR THREE PROGRAMS.

Programs	Num of APIs	Time	Explored paths	Encoded contexts
Curl	1,429	1h32m5.609s	1,581,679	127,029 (8.03%)
Httpd	3,311	1h45m10.820s	3,889,204	460,590 (11.84%)
OpenSSL	8.842	4h6m26.661s	5,550,546	1,007,124 (18.14%)
Total	13,582	7h23m43.090s	11,021,429	1,594,743 (14.47%)

In Table VII, we can see that APICAD totally explores 11,021,429 paths and a few explored paths (14.47%) can satisfy path constraints. As a result, 1,594,743 encoded contexts are finally generated for 13,582 API functions of these three programs in about 7h, i.e., ∼16.7 milliseconds to generate one encoded context. Take a heavily used API *CRYPTO_malloc* which occurs 455 times in OpenSSL as a concrete example, APICAD takes 2m30.926s to generate 7,055 encoded contexts for it, while ARBITRAR takes 3m12.055s (1.3x compared to APICAD) but only generates 5,970 valid contexts for it.

Similarly, analyzing a common API in Linux kernel also finishes within several minutes. For example, APICAD takes 11 minutes to analyze a widely used API *kzalloc()* with 29,133 contexts generated. Though it is feasible to analyze the whole Linux kernel with enough resources, we suggest that end-users select APIs by considering statistics [38] such as the number of occurrences or sensitive words in the name, and then analyze filtered APIs with APICAD. Otherwise, it can consume a lot of unnecessary time and resources on many inessential functions because Linux kernel is indeed a behemoth.

2) Document analysis: Document analysis does not take much time in this work since we do not need to label data or train models. The documents in our evaluation are collected from the website within seconds, while it takes about 215 seconds to preprocess them into parts corresponding to each API. After preprocessing, there are 6,170 functions to be analyzed in total, and we spend 728 seconds filtering the sentences and extracting semantics from them. Finally, the specifications for 2,824 functions are constructed.

To measure the precision and recall of the constructed specifications from documents, the ground truth is required. Because manual examination of all documents is practically impossible, we randomly sampled 600 functions (around 10% of documented functions) as the ground truth to inspect our obtained results. According to our manual retrieval, 329 of the sampled 600 functions have the usage directive sentences, which can construct 295, 19, and 54 specifications for return, arguments, and causality, respectively. Overall, APICAD generates 239, 13, and 36 specifications for return, arguments, and causality of 264 APIs. Out of these specifications, 216, 11, and 34 turn out to be real, giving rise to a precision of 90.6%. Considering 107 false negatives, we get a recall of 70.9%. These results show that our method is acceptable in extracting the usage semantics from documents.

F. Case study

In this section, we take one case to show the path exploration capability of APICAD and another two cases to further demonstrate the core insight of APICAD is reasonable, i.e., the specifications from code and documents can be complementary. All of these are from APIMU4C Project-Case benchmark, and APICAD can successfully detect them.

```
static int ssl3_generate_key_block(SSL *s, unsigned char *km, int num)
 1    unsigned char buf[16];
 2    k = 0;
 3    m5 = EVP_MD_CTX_new();
 4    s1 = EVP_MD_CTX_new();
      ……
 5    for (i = 0; (int)i < num; i += MD5_DIGEST_LENGTH) {
 6        k++;
 7        if (k > sizeof(buf)) {
 8            return 0; // goto err;
 9        }
          ……
10    }
      ……
11  err:
12     EVP_MD_CTX_free(m5);
13     EVP_MD_CTX_free(s1);
```

Fig. 5. A simplified code snippet in the Project-Case benchmark.

1) Valid path exploration: As shown in Figure 5, memory is allocated for *m5* and *s1* before entering the loop, but the memory is not properly released when the caller returns at line 8. To detect this bug, we need to reach line 8 and report that there is no *EVP_MD_CTX_free()* after *EVP_MD_CTX_new()* on the path. When entering the loop, k is known to be 0. Typically, it needs more than 16 cycles to meet the condition at line 7. However, the previous works [16], [18] only unroll the loop once and ignore the impact of loop variables. For such cases, though we still unroll the loop once, we replace the result of the binary operations related to loop variables with a new symbol, i.e., k becomes a symbolic value instead of a const, so the path condition at line 7 can be satisfied.

2) Bug detected by the specification from code: In OpenSSL of APIMU4C, there are three bugs related to lacking a check for the return value of *PACKET_buf_init()*. APICAD detects these three bugs based on the specification mined from code since *PACKET_buf_init()* does not have its document. Specifically, APICAD generates 1,174 traces for *PACKET_buf_init()* in total and the proportion of $ret.checked = true$ is 87.1%, which is above the threshold, so the

specification $ret.need\_check$ is considered valid. It should be noted that there are actually many problematic traces, but they are merged during detection since they are in the same location and have the same violation information. Finally, three bugs about *PACKET_buf_init()* corresponding to the bugs in APIMU4C are successfully reported by APICAD.

3) Bug detected by the specification from documents: *DH_new()* is a memory allocation function that allocates and initializes a *DH* structure. In Httpd of APIMU4C, *DH_new()* is only used at one place in httpd/modules/ssl/ssl_engine_init.c, however, the location is missing a check for its return value. On the one hand, $ret.checked$ of all the traces are $false$, so we cannot infer the correct usage of the return checking for *DH_new()* from code whatever the threshold λ is. On the other hand, APICAD filters out the description about the return of *DH_new()*: *"If the allocation fails, DH_new() returns NULL and sets an error code that can be obtained by ERR_get_error(3). Otherwise it returns a pointer to the newly allocated structure."*. By analyzing it, we acquire the return specification: "ret": {"value": [0], "cond": ["fail"]}. With this, the above defect is finally detected by APICAD.

IV. Discussion

In this section, we will discuss the limitations of our approach and possible ways to mitigate them.

(1) Imprecise coverage of all feasible paths. On the one hand, the *slice depth* controls the analysis scope for each API invocation, and many indirect calls, such as callback functions, are ignored to process, which can miss some pre- or post-conditions. On the other hand, we use conservative pointer analysis to identify *usage-related* functions and the path conditions are under-constrained, which can bring false positives. To precisely cover more feasible paths, some advanced points-to analysis approaches [39]–[41] can help APICAD resolve indirect calls and identify *usage-related* functions. In addition, we consider enabling chopped symbolic execution [42] to automatically exclude irrelevant functions by resolving their side effects but not directly ignoring them.

(2) Inaccurate document analysis. On the one hand, we choose keywords to filter the *usage-related* sentences. Though this is simple and practical in the real world, it does sacrifice some accuracy. On the other hand, the pre-trained model of HanLP [35] has shown excellent results on multiple datasets. However, it cannot eliminate inaccurate cases, especially without appropriate tuning for specific corpus. Equipping with advanced techniques and creating specific datasets is a feasible way to improve the performance of different NLP models on such jobs, and we will attempt to do further work on it.

(3) Restricted assumption. Though we utilize several methods to mitigate the demand for a large-scale and high-quality codebase, we are still limited by the codebase, especially when documents are absent. As discussed in previous works, there are at least two ways to mitigate the false cases brought by the restricted assumption: (a) Self iteration. Because all the analyzed features are recorded, we can gradually reduce the dependence on the size and quality of a codebase by iterating over the past analyzed features. (b) Get feedback from users. With user feedback [43]–[45], we can more easily rank the cases and reduce false positives to save users' effort in validating the final reports. Besides, as the core insight of recent works [18], [46], active learning can also be a feasible way to utilize user feedback for detection.

V. Related Work

In addition to works focusing on generic API misuse types, there are many works only for specific applications or a separate type of API misuse, such as cryptographic API misuse [2], [47], [48], web API misuse [49], [50], improper error handling [51]–[55] and improper causal handling [56]–[58]. Regardless though, constructing API specifications or usage violation rules is the cornerstone of API misuse detection.

The API specifications or usage violation rules can be manually acquired. For example, SSLint [2] takes predefined rules to find API misuse specific to SSL/TLS, IMChecker [9] leverages the rules written in Yaml [59] to define API specifications, and Semmle [10] detects API misuse based on correct or incorrect usage patterns written in CodeQL [60]. In addition, encoding hard-coded rules into checkers of static tools like ClangSA [12] is also a common way. Unlike such techniques, APICAD can acquire three types of specifications without much manual effort, but to make it more practical, we also consider enabling users to provide assistance in the future.

To relieve manual efforts, many techniques explore mining API specifications from code. As a typical work, APISAN [16] regards various majority usage patterns as specifications. It is generic without requiring manual efforts, however, it could bring a high false positive rate and fails to handle complex contexts [8], [18]. There are other techniques for automatically generating specifications from code, such as Nar-miner [61], JUXTA [62], JIGSAW [63], APIMiner [64] and PRMiner [65], which also require a large corpus. Though APICAD still follows the assumption that the majority usage is correct when generating code-based specifications, it can mitigate the negative impact of this assumption if the specifications from documents are available, as demonstrated in Section III-C. Besides, to improve the quality of code-based specifications in practice, we optimize the capability of path exploration for better utilizing the codebase and adjust the determination of "majority" according to the size of the codebase.

With the development of NLP, many works attempt to extract specifications from the texts in human language. For instance, Jdoctor [66] extracts executable procedure specifications from comments to help generate test cases. ICON [21] identifies the formal temporal constraints from documents to infer API causal relationships. Similarly, we specify to analyze the three types of usage in documents with the help of PoS tagging and dependency parsing. To retrieve the usage-related descriptions, iComment [67] uses the keywords such as "must" or "need", and some works [20], [68] enable the approaches of regex or shallow parsing templates. Moreover, Advance [19] trains a bidirectional GRU model with attention for classification by labeling multiple sentences, which achieves better

accuracy. In this work, we filter sentences via the keyword-based method. Although this results in an acceptable precision, it can also bring many false negatives [19]. When there is enough labeled training data, we can also leverage advanced techniques such as Advance to improve APICAD.

Besides, since API misuse bugs usually have security implications, Catcher [69] aims to generate test cases that result in crashes through search-based testing and static exception propagation analysis. In addition, given the location of an API misuse bug, some techniques, such as directed fuzzing [70] and fuzz driver generation [71], [72], can help produce specific test cases to further verify its security impact.

VI. Conclusion

In this paper, we presented APICAD, which analyzes code and documents to construct more comprehensive specifications for finding API misuse bugs in C/C++ programs. We evaluated the prototype of APICAD on a bug benchmark and several widely-used real-world programs. The results show that APICAD is effective in capturing diverse contexts for API invocations to deal with different types of API misuse bugs. In the future, we believe the combined specification will play a more significant role after applying more advanced techniques to code and document analysis.

Acknowledgments

We appreciate the anonymous reviewers for their valuable comments and suggestions to improve our paper. We also want to express our gratitude to Ziyang Li for developing ARBITRAR's symbolic execution engine, because part of APICAD is built on it. This work is partly supported by National Natural Science Foundation of China under Grant No.62172305.

References

[1] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 38–49.

[2] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting ssl usage in applications with sslint," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 519–534.

[3] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 602–613.

[4] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 289–305.

[5] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 886–896.

[6] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do java developers struggle with cryptography apis?" in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 935–946.

[7] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 392–403.

[8] 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. IEEE, 2019, pp. 11–20.

[9] Z. Gu, J. Wu, C. Li, M. Zhou, Y. Jiang, M. Gu, and J. Sun, "Vetting api usages in c programs with imchecker," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 91–94.

[10] "Semmle - Code Analysis Platform for Securing Software," 2021, https://semmle.com/.

[11] "Cppcheck - A tool for static C/C++ code analysis," 2021, https://cppcheck.sourceforge.io/.

[12] "Clang Static Analyzer," 2021, https://clang-analyzer.llvm.org/.

[13] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 240–250.

[14] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 35–44.

[15] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Automated Software Engineering*, vol. 18, no. 3, pp. 263–292, 2011.

[16] 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.

[17] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static api-misuse detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 265–275.

[18] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, "Arbitrar: User-guided api misuse detection," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1400–1415.

[19] 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*, ser. CCS '20, 2020, p. 1837–1852.

[20] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 815–825.

[21] R. Pandita, K. Taneja, L. Williams, and T. Tung, "Icon: Inferring temporal constraints from natural language api descriptions," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 378–388.

[22] 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.

[23] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 925–936.

[24] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 643–652.

[25] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 33–42.

[26] A. Head, C. Sadowski, E. Murphy-Hill, and A. Knight, "When not to comment: questions and tradeoffs with api documentation for c++ projects," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 643–653.

[27] "OpenSSL - Cryptography and SSL/TLS Toolkit," 2021, https://www.openssl.org/.

[28] "Functions in libcrypto without documentation," 2021, https://github.com/openssl/openssl/blob/master/util/missingcrypto.txt.

[29] D. Engler and D. Dunbar, "Under-constrained execution: making automatic code destruction easy and scalable," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 1–4.

[30] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 49–64.

[31] "GNU LIB C - Memory Protection," 2021, https://www.gnu.org/software/libc/manual/html_node/Memory-Protection.html.

[32] "curl - command line tool and library for transferring data with URLs," 2021, https://curl.se/.

[33] "httpd - Apache HTTP server project," 2021, https://httpd.apache.org/.

[34] "The Linux Kernel Archives," 2021, https://www.kernel.org/.

[35] H. He and J. D. Choi, "The stem cell hypothesis: Dilemma behind multi-task learning with transformer encoders," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2021, pp. 5555–5577.

[36] "Juliet test suite," 2021, https://samate.nist.gov/SRD/testsuite.php.

[37] "Static analysis benchmarks from toyota itc," 2021, https://github.com/regehr/itc-benchmarks.

[38] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[39] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[40] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1749–1766.

[41] D. Trabish, T. Kapus, N. Rinetzky, and C. Cadar, "Past-sensitive pointer analysis for symbolic execution," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 197–208.

[42] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 350–360.

[43] K. Heo, M. Raghothaman, X. Si, and M. Naik, "Continuously reasoning about programs using differential bayesian inference," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 561–575.

[44] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 462–473.

[45] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using bayesian inference," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 722–735.

[46] H. J. Kang and D. Lo, "Active learning of discriminative subgraph patterns for api misuse detection," *IEEE Transactions on Software Engineering*, 2021.

[47] P. L. Gorski, L. L. Iacono, D. Wermke, C. Stransky, S. Möller, Y. Acar, and S. Fahl, "Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic {API} misuse," in *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, 2018, pp. 265–281.

[48] I. Muslukhov, Y. Boshmaf, and K. Beznosov, "Source attribution of cryptographic api misuse in android applications," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 133–146.

[49] E. Wittern, A. T. Ying, Y. Zheng, J. Dolby, and J. A. Laredo, "Statically checking web api requests in javascript," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 244–254.

[50] S. Bae, H. Cho, I. Lim, and S. Ryu, "Safewapi: Web api misuse detector for web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 507–517.

[51] M. Acharya and T. Xie, "Mining api error-handling specifications from source code," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 370–384.

[52] C. Li, M. Zhou, Z. Gu, M. Gu, and H. Zhang, "Ares: Inferring error specifications through static analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1174–1177.

[53] 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.

[54] K. Lu, A. Pakki, and Q. Wu, "Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.

[55] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 499–510.

[56] P. Bian, B. Liang, J. Huang, W. Shi, X. Wang, and J. Zhang, "Sinkfinder: harvesting hundreds of unknown interesting function pairs with just one seed," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1101–1113.

[57] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *The 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*, 2021.

[58] H.-Q. Liu, J.-J. Bai, Y.-P. Wang, Z. Bian, and S.-M. Hu, "Pairminer: mining for paired functions in kernel extensions," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 93–101.

[59] "YAML: YAML Ain't Markup Language," 2021, https://yaml.org/.

[60] P. Avgustinov, O. De Moor, M. P. Jones, and M. Schäfer, "Ql: Object-oriented queries on relational data," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[61] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai, "Nar-miner: discovering negative association rules from code for bug detection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 411–422.

[62] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 361–377.

[63] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "{JIGSAW}: Protecting resource access by inferring programmer expectations," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 973–988.

[64] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 25–34.

[65] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

[66] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 242–253.

[67] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 145–158.

[68] Y. Chen, L. Xing, Y. Qin, X. Liao, X. Wang, K. Chen, and W. Zou, "Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 747–764.

[69] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, "Effective and efficient api misuse detection via exception propagation and search-based testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 192–203.

[70] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[71] 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*, 2019, pp. 975–985.

[72] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2271–2287.