RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection

Tao Lv[1,2], Ruishi Li[1,2], Yi Yang[1,2], Kai Chen[1,2,*]
Xiaojing Liao[3], XiaoFeng Wang[3], Peiwei Hu[1,2], Luyi Xing[3]
[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Luddy School of Informatics, Computing and Engineering, Indiana University Bloomington
{lvtao,liruishi,yangyi,chenkai,hupeiwei}@iie.ac.cn,{xliao,xw7,luyixing}@indiana.edu

Abstract

To use library APIs, a developer is supposed to follow guidance and respect some constraints, which we call *integration assumptions* (*IAs*). Violations of these assumptions can have serious consequences, introducing security-critical flaws such as use-after-free, NULL-dereference, and authentication errors. Analyzing a program for compliance with IAs involves significant effort and needs to be automated. A promising direction is to automatically recover IAs from a library document using Natural Language Processing (NLP) and then verify their consistency with the ways APIs are used in a program through code analysis. However, a practical solution along this line needs to overcome several key challenges, particularly the discovery of IAs from loosely formatted documents and interpretation of their informal descriptions to identify complicated constraints (e.g., data-/control-flow relations between different APIs).

In this paper, we present a new technique for automated assumption discovery and verification derivation from library documents. Our approach, called *Advance*, utilizes a suite of innovations to address those challenges. More specifically, we leverage the observation that IAs tend to express a strong sentiment in emphasizing the importance of a constraint, particularly those security-critical, and utilize a new sentiment analysis model to accurately recover them from loosely formatted documents. These IAs are further processed to identify hidden references to APIs and parameters, through an embedding model, to identify the information-flow relations expected to be followed. Then our approach runs frequent subtree mining to discover the grammatical units in IA sentences that tend to indicate some categories of constraints that could have security implications. These components are mapped to verification code snippets organized in line with the IA sentence's grammatical structure, and can be assembled into verification code executed through CodeQL to discover misuses inside a program. We implemented this design and evaluated it on 5 popular libraries (OpenSSL, SQLite, libpcap, libdbus and libxml2) and 39 real-world applications. Our analysis discovered 193 API misuses, including 139 flaws never reported before.

[*]Corresponding author.

CCS Concepts

• Security and privacy → Vulnerability scanners; **• Software and its engineering → Software safety**.

Keywords

API misuse; Integration assumption; Verification code generation; Documentation analysis

ACM Reference Format:
Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, Luyi Xing. 2020. 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 (CCS '20).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3423360

1 Introduction

Today's software is more composed than built, with existing program components, even online services, being extensively reused, often through integration of their Application Programming Interfaces (APIs). These APIs may only operate as expected under some constraints (referred to as *integration assumptions* or *IA* in our research), on their inputs (called *pre-conditions*, like a limit on parameter lengths), outputs (called *post-conditions*, e.g., "*return 1 or 0*") and invocation context (called *context conditions*, such as "*must be called from the same thread*"), which their users should be aware of. Although these IAs are typically elaborated in library documentation, they may not be strictly followed by software developers, as reported by prior studies [42, 44]. This affects the functionality of the software using these APIs, and often comes with serious security or privacy implications. As an example, `setuid`, which temporarily carries a high privilege, is expected to drop its privilege when its task is done, and failing to do so needs to be checked, as indicated in the `libstdc` documentation "*it is a grave security error to omit to check for a failure return from setuid()*"; this instruction, however, has not been respected by PulseAudio 0.9.8, leading to a local privilege escalation risk (CVE-2008-0008).

The fundamental cause of API misuse is that developers do not follow the IAs of APIs in documents, which therefore becomes the key sources for API misuse discovery. Prior research on API misuse detection and API specification discovery [21, 30, 37, 41, 52] rely on analyzing a set of code to infer putative IAs, so as to identify API misuses. However, these IAs are often less accurate, introducing significant false positives. In the meantime, the limited coverage of the code set can also cause many real-world misuse cases to fall through the cracks (Section 5.3).

Finding misuse from doc: challenges. Alternatively, one can recover their IAs from documentation for a compliance check on the software integrating them, which leverages more semantic information and therefore can be more accurate. Given the size of today's library documents (e.g., 327 pages for OpenSSL), manual inspection becomes less realistic. Prior research seeks to automate this process by inferring these assumptions using natural language processing (NLP) [42, 44, 47]. Particularly, it has been shown that for the well-formatted C# documents, a set of templates that specify the relations between subject and object are adequate in recovering IAs. This simple approach, however, no longer works well on less organized documents, like those for most C libraries, such as *OpenSSL*, where IA descriptions can be hard to fit into any fixed templates: as an example, we ran the code released by the prior research on *OpenSSL* and only observed accuracy of 49% (Section 5.3). How to effectively identify the API constraints from such documents turns out to be nontrivial.

Also the challenge is to automatically analyze each IA and determine a program's compliance with the assumption. The prior research maps a set of predicates on pre- and post-conditions to formal expressions (e.g., "*greater*" to ">") [44]. Less known is how these predicates are selected and whether they are representative. Most concerning is the missing of semantic analysis on context conditions (except on API order [42, 54]), which can describe the information flow relations (between calls or between the caller and the callee) that must be established to avoid security or privacy fallouts. For example, the libpcap document describes how a program should use two APIs `pcap_close` and `pcap_geterr`: "*you must use or copy the string before closing the pcap_t*". Here, the references to `pcap_geterr` and `pcap_close` are implicit, through natural language description ("*close*" and "*pcap_t*"), which specifies information-flow connections: one needs to use the output of `pcap_geterr` ("*string*" in the IA) – a data flow, before running `pcap_close` to terminate the handler `pcap_t` – a control flow. Violation of this IA could lead to a use-after-free bug (Figure 2). Discovery and interpretation of such relations are by no means trivial.

Given the challenges in automatic IA discovery and semantic analysis on loosely organized library documents for the API compliance check, so far we are not aware of any end-to-end solution that has been applied to real-world software and led to the discovery of security-critical bugs, particularly those never reported before.

Advance. Recent years have witnessed significant progress made in machine learning and NLP (e.g., Gated recurrent unit (GRU) and Attention) techniques, in terms of performance and effectiveness, which potentially offers new means to address the challenges in API misuse detection from library documentation. In this paper, we report a new attempt to innovate over these new techniques, which moves the state-of-the-art of IA discovery and compliance check a step forward, enabling detection of security-critical API misuse from real-world applications. Our technique, dubbed *Advance* (assumption discovery and verification derivation from the document), is capable of automatically analyzing less organized C library documentation to recover and interpret IAs and then translating them into verification code executed by CodeQL [4] to find the API misuses in a program's API integration. For this purpose, Advance is designed to overcome the technical barriers mentioned earlier, based upon several key observations. More specifically, we found that with the diversity in the ways IAs are described, the related sentences are characterized by a strong sentiment to emphasize the constraints that API users are expected to follow: e.g., "*the application must finalize every prepared statement*"; "*make sure that you explicitly check for PCAP_ERROR*"; "*it is the job of the callback to store information about the state of the last call*". Such sentiments cannot be easily described by templates but can be captured using *semantic analysis*. In our research, we utilized a *Hierarchical Attention Network* (*HAN*) [51] to capture the sentences carrying such a sentiment when the entities (API names, parameters) under the constraint (as expressed by the sentiment) can be controlled by the API caller (Section 3.2). This has been done by training a classifier named S-HAN on 5,186 annotated data items, which is shown to work effectively in inferring IA sentences from library documents.

Also discovered in our research is the pervasiveness of similar IA components within a document and in some cases, across different libraries. These components imply common assumption (constraint) types for certain kinds of operations, which are often among the most important and can be security critical. For example, "*... is not threadsafe*", an expression describing thread safety; "*... must be freed by...*", an IA asking for resource release – an operation with a significant security implication. Leveraging this observation, our approach automatically parses IA sentences into dependency trees and runs frequent subtree mining [53] to discover the components (called *Code Descriptions* or *CDs*, which are the smallest textual description units that can be mapped to verification code snippets.) for such common assumption types. Each of such assumptions is then mapped to a *verification code snippet* (*VCS*) for building up the complete *verification code* (*VC*), which may contain multiple VCSes (Section 3.4). Our research shows that the discovered assumption types cover 75% of CDs in the IAs (the column "*CD-Cov*" in Table 2).

To discover the context condition from an IA sentence, we capitalize on the observation that although the reference to an API or its parameters can be implicit, its semantics must come close to the description of its functionalities. For example, "*closing the pcap_t*" represents the API `pcap_close` and `pcap_t` is from the first parameter of `pcap_geterr`. Therefore, we trained a sentence embedding model in our research to convert the components of each IA sentence, as identified through shadow parsing, into vectors, so as to compare their similarity with those generated from the descriptions of different APIs (Section 3.2). In this way, not only can we discover the implicit reference to APIs and parameters, but we can also find out the data-flow (through parameters) and control-flow (through invocations) relations among APIs that form assumption types.

Discoveries. In our research, we implemented Advance on StanfordCoreNLP [38], AllenNLP [26], TREEMINER [53] and CodeQL [4], and further evaluated our prototype on the documentations of 5 popular libraries, including OpenSSL, SQLite, libpcap, libdbus and libxml2. Our study shows that Advance can effectively identify IA sentences, achieving an accuracy of 88% (on labeled dataset). Also, our prototype successfully recovered over 69% of IAs.

Further running our prototype on 39 applications integrating these libraries, including popular programs like wireshark, openvpn and ettercap (Section 5.4), our approach detected 193 instances of API misuse, including 139 problems never reported before. So far 16 of them have already been confirmed by the application developers. Many of these cases are security-relevant, with 6 of them documented by CVEs. These newly discovered API misuses can lead to system crash (NULL-dereference), DoS attacks or information leakage (improper resource shutdown or release) . As far as we know, this is the first time that an NLP-based end-to-end API misuse system automatically captures new security bugs by analyzing loosely organized documents. We will release the first version of Advance through Github later[1].

Contributions. The contributions of this paper are summarized as follows:

• *New technique.* We developed a novel technique for automatic, end-to-end IA discovery and verification code derivation. Our approach addresses several key technical barriers that prior research fails to overcome, including the use of sentiment analysis to recover assumptions from loosely organized library documents, subtree mining to identify the common operations related to the integration assumption, and sentence embedding to detect implicit description of information flows. These innovations enable a new end-to-end, document-to-code analysis capability and contribute to the advance of scientific research in this area.

• *Implementation and discoveries.* We implemented our technique and open-sourced our prototype. The evaluation of our prototype on popular libraries and real-world applications leads to the discovery of 193 API misuses (most with significant security or privacy implications), including 139 bugs never reported before. Such discoveries have never been made automatically on loosely organized API documents before, up to our knowledge.

2 Background

2.1 API Misuse

Software libraries often come with documentation that describes how to use their APIs properly, under constraints. Failure to follow such guidance can lead to API misuses, breaking the constraints on API inputs (e.g., overlong input parameters), the usage of outputs (e.g., unchecked return value), and the context for API invocations (e.g., inverse invocation sequence). Such misuses can have serious consequences, affecting the functionality of the software integrating these APIs, often with security or privacy implications, such as memory leaks, denial of service through application crash, etc. For example, most libraries require the application integrating their APIs to free allocated variables after its whole lifetime, which is usually specified in the API documentation. If the application fails to do so, a bug can be introduced (CWE-401: Missing Release of Memory after Effective Lifetime), which will cause memory over-consumption, resulting in a denial of service (by crashing or hanging the program). Despite the fact that such security-critical constraints are described in the documentation, API misuse is still one prevalent cause of software bugs [35].

[1]The release information is available at: http://kaichen.org/tools/Advance.html

2.2 NLP primitives

To analyze the library documentations, we leveraged a set of NLP techniques in our research. They are briefly introduced here.

Dependency parsing. Dependency parsing is an NLP technique to analyze the grammatical relations between the linguistic units (words) in one sentence and extract the syntactic structure of this sentence. As illustrated in Figure 4(a), in a dependency tree, the verb of a clause structure is the root and the other linguistic units are nodes linked by grammatical relations to the root. According to the Stanford Parser manual [1], there exist approximately 50 grammatical relations. In our research, we utilize the AllenNLP parser [26] to generate dependency trees for mining CDs.

Word embedding. A word embedding $W : words \rightarrow V^n$ is a parameterized function mapping words to vectors (200 to 500 dimensions), e.g., $W(\text{“}education\text{”}) = (0.2, -0.4, 0.7, ...)$, which captures a word's relation with other words in its context. Among the word embedding methods, word2vec [40] is a state-of-the-art and computationally efficient technique. Such a mapping can be done in different ways, e.g., using the continual bag-of-words model and the skip-gram technique to analyze the context in which the words show up. Such a vector representation ensures that synonyms are given similar vectors and antonyms are mapped to dissimilar vectors. In our study, we utilized word2vec to generate semantic word vectors from library documentation for the sentiment analysis.

Sentiment analysis. Sentiment analysis, also known as opinion mining, is a technique using NLP and text analysis to identify, extract, and study the opinion and subjective information. One of its main tasks is sentiment classification, which aims to classify the polarity of the text into positive or negative opinions. Recent research focuses on leveraging Deep Learning (DL) classifiers for sentiment classification. Popular off-the-shelf models include Text-CNN [32], RCNN [34], and HAN [51]. In our research, we compared the effectiveness of the aforementioned models and utilized a variance of HAN in discovering the sentences with IA (Section 3.2).

3 Advance: Design

In this section, we elaborate on the design and implementation of Advance – our technique for automatic assumption discovery and verification code derivation. We first give an overview of the whole design, its architecture and an example that shows how it works, and then move onto its individual components.

3.1 Overview

Architecture. Figure 1 illustrates the architecture of *Advance*, including three key components: *IA discovery*, *IA dereference* and

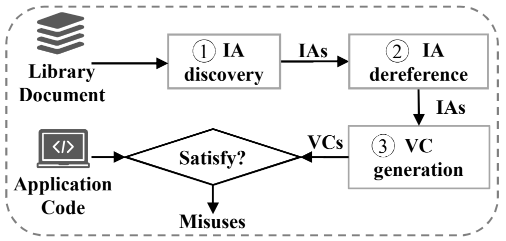


Figure 1: Architecture of *Advance*

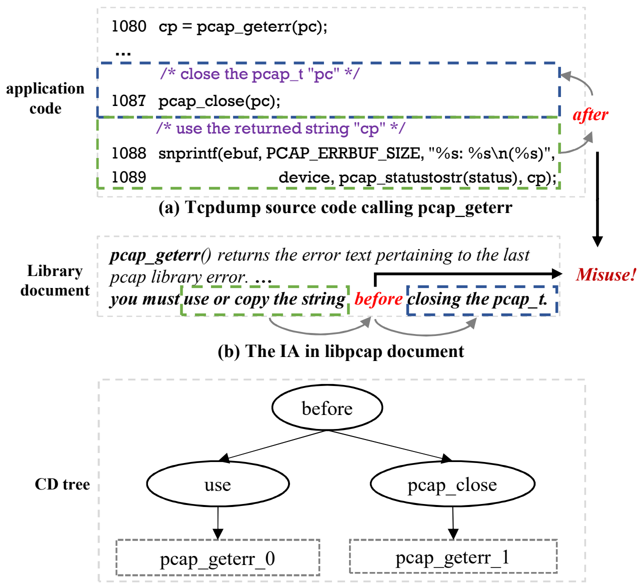
VC generation, together with its workflow. Specifically, at step ①, IAs are automatically discovered from the loosely structured API descriptions in the documents, through an S-HAN-based classifier that identifies IAs from the sentiments in the text. Then for each IA, Advance replaces its implicit references to APIs or their parameters with explicit ones (called IA dereference, step ②). For this purpose, our approach utilizes semantic analysis and lexical analysis to map from the descriptions to their corresponding APIs/parameters. Finally, after identifying the code descriptions (CDs) in IAs and their context conditions, Advance generates *CD trees* to organize those CDs, which are utilized to produce verification code (VC) under the context (step ③). The generated VC can then be executed by CodeQL to find the bugs in a program's API integration.

Example. Figure 2 explains how Advance works through an example. The code snippet in Figure 2(a) is extracted from a popular application named tcpdump, which utilizes two APIs from libpcap: `pcap_geterr`, `pcap_close`. Here `pcap_geterr` is first used to get the information about network packet errors, and store it to the memory referenced by the returned pointer `cp` (of the handler `pcap_t`). `pcap_close` closes `pcap_t` and therefore releases the memory including the error message pointed to by `cp`. However, in Line 1088, tcpdump tries to print out the error message using the invalid pointer `cp`, which introduces a use-after-free bug.

Actually, the libpcap document describing `pcap_geterr` clearly states that "*you must use or copy the string before closing the pcap_t*" (Figure 2(b)). This guidance, however, has not been followed by the developers of tcpdump. Advance is able to detect this API misuse. More specifically, it first automatically identifies from the libpcap document the above sentence, which is considered to contain the IA based upon the sentiment ("*you must ...*"). However, the sentence cannot be directly translated into the verification code, as it is not clear what means by "*string*", "*pcap_t*" and "*closing*". To make sense of such description, Advance continues to perform IA dereference and find that "*string*" indicates the return value of `pcap_geterr`, *"closing"* refers to `pcap_close` and "*pcap_t*" is its parameter. By further analyzing the context conditions (i.e., "*before*"), Advance constructs the CD tree (Figure 2(c)) for VC generation. As shown in the CD tree, the return value of `pcap_geterr` (i.e., `cp`) should be used before `pcap_close` operates on the parameter of `pcap_geterr` (i.e., `pc`). This tree is used to generate the verification code based upon the following code snippets: "`argv2.getASuccessor+() = argv1`" for "*before*", "`Expr reach,definitionReaches(argv, reach)`" for "*use*", "`FunctionCall fc, LocalScopeVariable v, VariableAccess u, fc.getTarget().hasQualifiedName("argv1") and v.getAnAccess() = argv2 and u = v.getAnAccess() and fc.getAnArgument() = u`" for "*call with*". The code is then used to inspect tcpdump to find the misuse.

3.2 IA Discovery

As mentioned earlier, automatically extracting IAs from documents, especially those less structured, is nontrivial. Template or keyword based approaches, as proposed by previous studies [44], do not work well. Particularly, in the presence of the API documents from different developers, not conforming with any writing convention, finding templates to cover most IAs is found to be a mission impossible. A key observation in our research is that with the diversity of



```
1080 cp = pcap_geterr(pc);
...
     /* close the pcap_t "pc" */
1087 pcap_close(pc);
     /* use the returned string "cp" */
1088 snprintf(ebuf, PCAP_ERRBUF_SIZE, "%s: %s\n(%s)",
1089                      device, pcap_statustostr(status), cp);
```

(a) Tcpdump source code calling pcap_geterr

pcap_geterr() returns the error text pertaining to the last pcap library error. ... you must use or copy the string before closing the pcap_t.

(b) The IA in libpcap document

(c) CD tree, pcap_geterr_0 and pcap_geterr_1 represent the return value and first parameter of pcap_geterr respectively

Figure 2: An example of the API misuse

writing styles, all IAs properly presented in the documents are characterized by a strong sentiment to stress the constraints. Actually the more important the IAs are, the more forceful the descriptions would be. For example, the document of *SQLite* states "*the application must finalize every prepared statement*"; an IA in libpcap alerts the developer "*make sure that you explicitly check for PCAP_ERROR*". Based on these observations, our design of Advance utilizes sentiment analysis to capture these assumptions.

Sentiment-based IA classifier. Specifically, we utilize the Bi-GRU-based encoder [24] and an attention mechanism [49] to discover IAs: Bi-GRU-based encoder is suitable for learning the context of one sentence to generate a representation, while the attention mechanism focuses the model more on sentimental words, which developers often use in IAs to make sure the APIs are correctly used. Particularly, inspired by the Hierarchical Attention Networks (HAN) [51], one of the most popular models that integrate the Bi-GRU-based encoder and the attention mechanism, we design a new model called *Sentence-HAN*, or *S-HAN* for short, to extend the conventional HAN, which is meant to classify documentation, for sentence classification. Figure 7 in Appendix illustrates the design of S-HAN. Its bottom layer is the word encoder, which includes Bidirectional GRUs that get annotations of the words through collecting information from both directions of a sentence. The inputs of the encoder are the vectors of words w_i produced by an embedding model and its outputs are the word annotations h_i. Considering that different words do not contribute equally to the result of classification, an attention layer is added after the encoder to underline sentiment-related expressions: the Multilayer Perceptron (MLP) in the attention layer receives u_i to output the attention weight a_i through the softmax function. Finally, word annotation vectors h_i are summed based upon the attention weight a_i into a sentence vector v, i.e., $v = \sum_{i=1}^{T} a_i h_i$, and the vector is used for classification through a softmax function.

Since there is no open dataset for training our model, we manually collected and annotated 2,601 IAs (1,296 IAs are from back-translation) and 3,881 non-IA from OpenSSL documentation. Since IAs only appear on a small set of sentences, we utilized the back-translation [25] to augment the dataset: by translating a sentence in English to another kind of language (e.g., Spanish) and then translating it back, we could get more sentences with similar meanings. Then these sentences could be added to our dataset. In our evaluation, S-HAN achieved an accuracy of 88% in discovering IAs (Section 5.2), more accurate than other models (e.g., Text-CNN and RCNN). Also, from the attention layer, we observed the words in a sentence that have significant impacts on the classification results. For example, in the sentence "*It is the caller's responsibility to free this memory with a subsequent call to OPENSSL_free*", the word "*responsibility*" reflects a strong sentiment, indicating an IA being communicated, which is in line with what we see from the documents.

3.3 IA Dereference

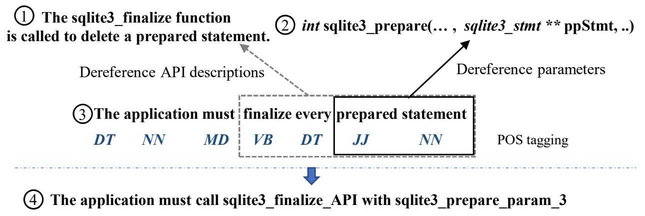
To interpret a discovered IA, oftentimes we need to identify its implicit references to an API name or parameters. For example, in the IA "*The application must finalize every prepared statement*", "*finalize every prepared statement*" refers to the API `sqlite3_finalize` and "*prepared statement*" indicates the third parameter of the API `sqlite3_prepare`. These references are critical for understanding the information-flow relations between the caller and the API being called and between different API invocations. Without resolving them, an IA cannot be translated into the verification code. To address this problem, Advance utilizes existing tools, such as AllenNLP [26] and NeuralCoref [29], to eliminate anaphora. However, none of such techniques can address subtle implicit references, as those in the above example. Our solution is a semantic-based approach for API dereference and a lexical analysis for parameter discovery, as elaborated below.

```
Nall:{<NN.*|CD|LS.*|PRP$>}
Npre:{<DT|PDT|PRP|CD>}
VADV:{<RB.*>*<VP|VB.*>}
NP:{<JJ.*>*<Nall>+}
VP_passive:{<Nall>+<MD>?<VADV><RB.*>?<VADV>+}
VP_active :{<VADV><IN>*<Npre>*<RB.*>*<JJ.*|VBN|VBG>*<
    Nall>+}
```

Listing 1: Shallow parsing grammars

Semantics-based API dereference. Our dereference solution is based upon the observation that an implicit API reference should be semantically similar to the descriptions of the API's functionalities. This allows us to compare their semantic meaning to identify those closely-related pairs. To this end, Advance performs efficient NLP analyses such as a shadow parsing to recover these references from IAs and then analyze their semantics and that of API description through sentence embedding.

Specifically, an implicit reference to an API describes its operations, which typically contains verb. Therefore, to find these references, we utilize the Part-of-Speech (POS) tagging, and shallow parsing [46] to mark words in discovered IA sentences and recover all verbs. For example, in Figure 3, after parsing the IA "*The application must finalize every prepared statement*", the word "*finalize*" is recognized as a verb (tagged as "*VB*") and the word



① The sqlite3_finalize function is called to delete a prepared statement.
② int sqlite3_prepare(… , sqlite3_stmt ** ppStmt, ..)
Dereference API descriptions
Dereference parameters
③The application must
finalize every
prepared statement
DT NN MD VB DT JJ NN
POS tagging
④ The application must call sqlite3_finalize_API with sqlite3_prepare_param_3

Figure 3: Example of IA dereference process (API and parameter dereference). After POS tagging, IA (③) was parsed to recognized "*finalize every prepared statement*" as a verb phrase and "*prepared statement*" as a noun phrase. These phrases were then dereferenced to the API `sqlite3_finalize` (based on API functionality sentence ①) and the parameter `ppStmt` of `sqlite3_prepare` (based on API declaration ②), to obtain the dereferenced IA (④).

"*statement*" is a noun (tagged as "*NN*"), which are combined into a verb phrase (tagged as "*VP*"). To identify different types of verb phrases (e.g., VBN and VBP), our approach leverages a set of rules that describe these *VP*s based upon their POS (in Listing 1) and further utilizes regular expressions to apply the rules on the parsed sentences to captures the verb phrases. For example, *VP* includes active verb phrases (*VP_active*), such as "*finalize every prepared statement*", and passive verb phrases (*VP_passive*), like "*The string must be deallocated*". *VP_passive* is composed of at least a noun (i.e., *Nall*) and a verb (i.e., *VADV*). Sometimes, a modal verb (i.e., *MD*) may also exist between the noun and the verb, and more verbs may also be included. Listing 1 presents the details of these rules[2]. Note that some extracted verb phrases are API references, which are identified through comparison with API descriptions.

An API description is characterized by the appearance of the API name at the beginning of a paragraph. This allows us to extract the sentences from the paragraphs to compare their semantic meanings with that of the verb phrases discovered from IAs. In our research, we found that typically the first sentence of the API description explains its functionality, so it is picked out for the comparison. For this purpose, Advance utilizes sentence embedding to transform a sentence into a vector to represent its semantics. Here, the embedding model was trained in our research, without supervision, on each library. The semantic comparison is performed by calculating the cosine similarity between the vector for a verb phrase and that of the API functionality sentence (S_1 and S_2), i.e., $sim = \frac{S_1 \cdot S_2}{\|S_1\|\|S_2\|}$.

Looking at similar pairs, our approach captures an implicit API reference from a verb phrase and dereferences it using the closest API functionality sentence[3] in semantics to the phrase. Then we replace the reference with "*call #API with #noun_phrase*", where *#API* is the name of the API discovered, and *#noun_phrase* is the noun in the verb phrase ("*prepared statement*" in the example). As shown in Figure 3, the IA is transformed to "*call sqlite3_finalize_API with prepared statement*". In the absence of matched API descriptions (that is, low similarity across all sentences), a verb phrase is not considered to contain API reference. Our experiment shows this approach is quite effective, achieving an accuracy of 94% (Section 5.2).

[2]82 POS tags are given in the website: http://www.surdeanu.info/mihai/teaching/ista555-fall13/readings/PennTreebank.html.
[3]An API functionality sentence describes the API functionality.

Dereferencing parameters. Unlike the implicit API reference, a subtle indication of an API parameter in an IA has lexical connections to the name of the parameter, which is meant for a reader to easily locate the related code: we found in our research that in most cases, such a reference is in one of three forms – an abbreviation of a parameter name (e.g., `name` for "*zName*"), an extension of the name (e.g., using "*prepared statement*" to describe the parameter `ppStmt`), or the type of the parameter (e.g., `sqlite3_snapshot`). Based on this observation, our approach compares the semantics of a possible reference with a associated parameter's name and type to report the most similar pair.

Specifically, we first recover possible parameter references from IA sentences. Such references are often in the form of noun phrases (tagged as *NP*) that contain at least one noun (tagged as *NP*), sometimes with one or more adjectives (tagged as *JJ*) in front of the noun. This allows us to construct the rule of *NP* (see Listing 1) to detect such references. For API parameters, our approach directly extracts their names, descriptions, types from the API declarations using regular expressions. For example, in the declaration `int sqlite3_prepare(..., sqlite3_stmt** ppStmt, ...);`, `sqlite3_stmt**` is the type and `ppStmt` is the name. Then we use a regular expression to determine whether the relationship between the noun phrases (possible references) and the parameter names/-types can be characterized as abbreviations or expansions. The details are shown in Section 4.

Note that, in most cases, a dereferenced parameter appears in the API document at the locations close to the descriptions of the IA referring to it. So in the presence of multiple candidates (parameters apparently related to a noun phrase in an IA sentence), the one located closest to the IA is chosen to replace the implicit reference (the noun phrase), using its API name and parameter index (`API_param_idx`). For example, the parameter reference (in Figure 3 ③) is changed to `sqlite3_finalize_param_3` (in ④), representing the third parameter of `sqlite3_finalize`.

3.4 Verification Code Generation

After recovering IAs from documentation and resolving the implicit references, Advance is ready to generate verification code (VC) for finding API misuses in a program's API integration. As mentioned earlier, IAs are highly diverse, containing constraints on APIs' inputs (*pre-conditions*), outputs (*post-conditions*), and invocation context (*context conditions*), which is unlike the simple arithmetic and logic requirements handled by the prior research [44].

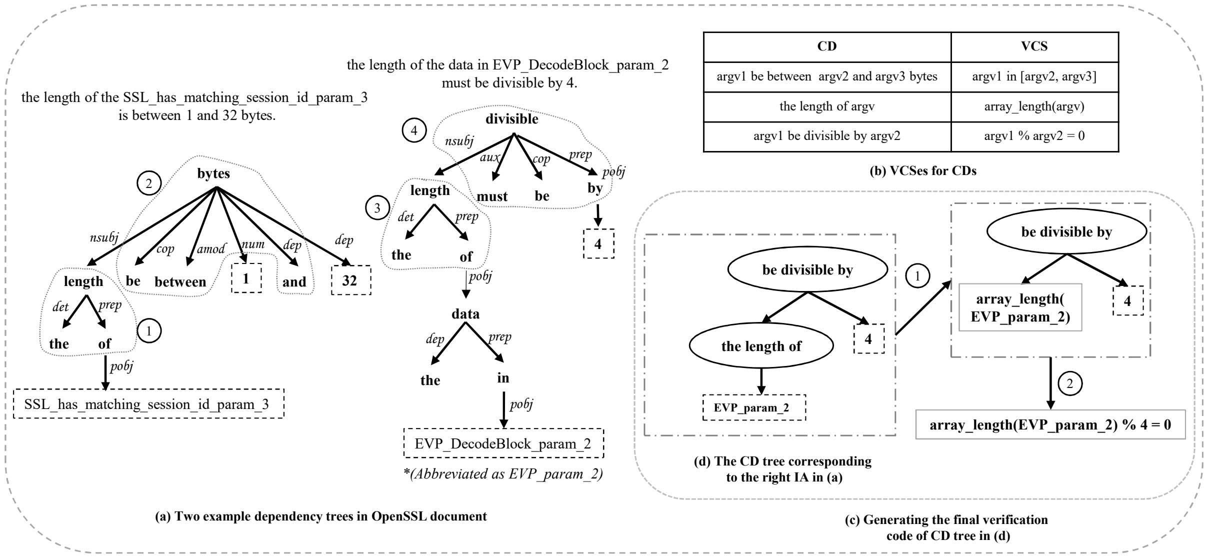
Automatic generation of proper VC even in the form of the verification tool queries is nontrivial. For example, data dependency and code sequential should be specified in VC, since otherwise the verification tool cannot perform the check correctly. One possible solution is to use machine learning to automatically synthesize the inspection code, which however requires a large amount of labeled data for model training that is unavailable for the problem of API misuse detection. Our solution is based upon an observation that important and often security-sensitive constraints tend to carry common components, not only within a document but also across different documents, indicating the presence of categories of critical conditions on API use one needs to follow. For example, "*the length of*" is a popular phrase in pre-conditions that indicates a type of constraints on the size of API parameters. Therefore, our idea is to break an IA into several small components (called Code Descriptions or CDs) and only map the most frequently used ones to verification code snippets (VCSes), which requires minimum manual effort. Then Advance automatically assembles these VCSes and parameterize them, based upon the combinations of their CDs in different IAs, to generate the complete VCs. During this process, the data and control dependencies among CDs are discovered through CD trees, and further preserved in the VC through traversal of these trees to link different VCSes together. Follow we elaborate this design.

Code description discovery. Ostensibly the discovery of the frequently used CDs can be done through a sliding window (N-grams) to find out the sentence fragments that show up several times in a document or across documents. This simple approach, however, does not work well on the analysis of IAs. The N-gram does not carry any syntactic and semantic information and can therefore cut into CDs and link less meaningful sentence fragments together: for example, "*data must*" (shown in Figure 4 (a)) will be extracted as a phrase of high frequency when the window size is 2; however it is not meaningful and does not provide any information about assumptions to be followed in API integration.

Therefore, Advance takes a syntax and semantics savvy solution, transforming an IA into a dependency tree and mining the most frequently used subtrees over its grammatical structure. An example of an IA's dependency tree is illustrated in Figure 4 (a). Here each node of the tree is a word, and different nodes are connected based upon their dependency type. For example, the word "*length*" and "*the*" are linked with the `det` type. Over such a tree, we run TREEMINER [53], an algorithm that discovers frequent subtrees, each of which describes a meaningful grammatical unit (template) such as a phrase. For example, Figure 4 (a) shows the dependency trees of two IAs, with four popular subtrees discovered across documents are circled with dotted lines and labeled (i.e., from 1 to 4), each being an automatically generated template. Note that we remove negative words (e.g., "*not*" and "*never*"), chronological words (e.g., "*before*") and modal verbs (e.g., "*should*") from dependency trees before mining, for detection of small meaningful units that can be easily extended or connected to other units through these words. As an example, from two descriptions "*be used*" and "*seldom be used*", only one CD is identified, since "*seldom be used*" can be automatically extended from the VCS of "*be used*". After that, for each subtree, we built a VCS (in the verification tool's query language) and store it in an *initial CD dataset*. For example, in Figure 4 (b), the CD "*the length of argv*" is converted to the VCS `array_length(argv)`.

In this way, each IA is then transformed into a dependency tree, whose subtrees are further compared across those of other IAs to find popular CDs. Our study shows that this approach can achieve an accuracy of 75% in detecting CDs (Section 5.2), indicating that most code descriptions can be captured by popular subtrees.

Verification code generation from CD. Given an IA extracted from a document, Advance first identifies its CDs by looking up the initial CD dataset, which as mentioned earlier, contains popular code description templates as discovered from frequent subtree mining within or across documents. Note that the manual translation of


the length of the SSL_has_matching_session_id_param_3 is between 1 and 32 bytes.
bytes
nsubj
cop
amod
num
dep
dep
length
be
between
1
and
32
det
prep
the
of
pobj
SSL_has_matching_session_id_param_3
the length of the data in EVP_DecodeBlock_param_2 must be divisible by 4.
divisible
nsubj
aux
cop
prep
pobj
length
must
be
by
4
det
prep
the
of
pobj
data
dep
prep
the
in
pobj
EVP_DecodeBlock_param_2
*(Abbreviated as EVP_param_2)
(a) Two example dependency trees in OpenSSL document
(b) VCSes for CDs
be divisible by
the length of
4
EVP_param_2
be divisible by
array_length(EVP_param_2)
4
array_length(EVP_param_2) % 4 = 0
(d) The CD tree corresponding to the right IA in (a)
(c) Generating the final verification code of CD tree in (d)

CD	VCS
argv1 be between argv2 and argv3 bytes	argv1 in [argv2, argv3]
the length of argv	array_length(argv)
argv1 be divisible by argv2	argv1 % argv2 = 0

Figure 4: An example of automatically generating verification code for checking API misuse.

the CDs in the initial CD dataset to VCSes is a one-time effort and not required to be done by Advance users. When a subtree c_i in the IA is found to match CD c_j in the dataset, our approach automatically generates the verification code snippet (VCS) of c_i (VCS_i) by transforming that of c_j (VCS_j). Note that VCS_i may not be identical to VCS_j, since at this point, we need to consider the impacts of the terms removed, including negative terms ("*not*", "*seldom*", etc.), chronological terms, modal words, etc., as mentioned earlier. The new snippet VCS_i therefore should be chosen from a variation of VCS_i in the dataset, according to the term presented. Also importantly, as a template, VCS_j or its variation needs to be parameterized with variables and constants such as number, string (see Figure 4 (a)) before it can be instantiated into VCS_i. For example, consider the IA "*the length of the data in EVP_DecodeBlock_param_2 must be divisible by 4*", with a popular CD "*the length of argv*", its VCS discovered from the initial CD dataset `array_length(argv)` need to be instantiated into `array_length(EVP_DecodeBlock_param_2)`.

After creating the VCSes for individual CDs in an IA, our approach links them together based upon their relations such as context condition, as described by the dependency tree. To this end, we convert the dependency tree to a *CD tree*, which models the relations between different verification code snippets (VCSes) as derived from the grammatical relations between their descriptions. Specifically, each leaf node of the CD tree is the entity representing variables or constants, and other nodes are CDs. Two CDs are connected if there exists a relation between them in the dependency tree. For example, Figure 4 (d) shows the CD tree of the IA "*the length of data in EVP_param_2 must be divisible by 4*" (in Figure 4 (a)); here the CDs "*the length of*" and "*be divisible by*" are linked together since "*length*" and "*divisible*" are also connected in the dependency tree (Figure 4 (a)). Also importantly, the directed edges in the CD tree mostly inherit the orientations of those in the dependency tree, with a CD related to a subtree. The exception is caused by the *temporal CD*, which specifies the sequential order between CDs (e.g., "*after*", "*before*"): in this case, the directed edge always starts from the temporal CD.

Over the CD tree, Advance generates the full verification code for the IA by traversing the tree, as presented by Algorithm 1 in Appendix. Specifically, our approach starts from the left-most leaf of the tree to retrieve its parent and siblings (Line 2-4), where the leaf nodes here are the parameters for the VCS template of their CD parent (Line 5-6). For example, in Figure 4 (c), the CD "*the length of*" has a leaf `EVP_param_2`; using the VCS template in Figure 4 (b), the generated VCS is `array_length(EVP_param_2)` (Step ①). This process continues until there is only the root node left in the CD tree. All the VCSes created (through parameterization) and connected during the traversal then form the IA's verification code. Again let us look at the example in Figure 4 (d): the VC produced is `array_length(EVP_param_2)% 4 = 0` (Step ②). In this way, all pre-/post-/context conditions can be correctly represented in the VC according to the description in the IA, and are ready for the verification tool to use to discover API misuses.

4 Implementation

We implemented a prototype of Advance in our research on top of a set of tools, as described in Table 3 in Appendix. Below we elaborate on the implementation details of each component.

IA discovery. We trained the Word2vec class of gensim [56] for 100 iterations to build up our own word2vec model, whose word vector was set to 300 (the commonly used value) and window size (maximum distance between the current and predicted word within a sentence) to 3. The training corpus of our Word2vec model was crawled from the Linux manual pages of library functions [17] (5.8MB with 40,000 sentences).

We utilized the default parameters of the word part of HAN [51] to train the S-HAN model, except for a customized setting for word embedding (dimension 300, batch size 16, and epoch 2). The backbone of the S-HAN model consists of a bidirectional layer (50 layers, regularization of L2 with a regularization factor of 1e-8), a dense layer (100 layers, ReLU activation function, the same regularization as the bidirectional layer), and an attention layer (1 layer, normal distribution initialization, supporting masking, and no regularization). Also, the categorical cross-entropy loss and Adam optimizer with learning rate 0.001 were used in the model training.

IA dereference. Since the traditional references are in the forms of pronouns, we combine several tools of anaphora resolution together, including NeuralCoref [29], AllenNLP [26] and StanfordCoreNLP [38]. As the accuracy of the tools is not high (e.g., 58% for Winograd Schema Challenge), we accept the resolution results only when the three tools come to the same result. In the process of dereferencing implicit API and parameter, we perform the shallow parsing using StanfordCoreNLP [38] to tag the POS of words in IAs. The threshold of *sim* is set to the similarity at 10% of the all the match result, which gives the highest accuracy according to our evaluation. We select the shorter one among the noun phrase and the API parameter name or type and then split them into characters. The split characters are joined with ".*" to form the regular expression. Then the regular expression is used to match to the longer one to decide whether a noun phrase is a abbreviation of expansion of the API parameter names or types.

Verification Code Generation. In our implementation, we utilized CodeQL [4], a popular code analysis engine capable of performing information-flow analysis, as a verification tool.

In the process of building the initial CD dataset, a CD may not be mined out even if it is commonly used. For example, the CD "*should be released*" and the CD "*should be freed*" express the same meaning, but use different words "*freed*" and "*released*". When Advance mines the most frequently used sub-tree, they are viewed as different CDs and may not be viewed as "frequently" used. To handle this problem, we automatically build a dictionary of synonyms using our trained Word2vec model and choose the most frequent word as the representative word for each synonym group. When a word in an IA appears in the dictionary, we replace the word with the representative word before mining. In this way, the CDs with the same meaning could be mined out.

Also, we view the subtree appearing more than 3 times (i.e., the parameter "minimum-support" equals 3/(the total number of IAs)) as the CD. We mine the CDs from the two libraries (i.e., OpenSSL and SQLite). In the evaluation, we find the mined CD can cover 75% of the CDs in the documents of other libraries. Details of the analysis are shown in Section 5.2.

5 Evaluation

In this section, we describe our evaluation on Advance, including the effectiveness of both its end-to-end operation and individual components, as well as its run time performance. After that, we compared Advance with static API misuse detectors [30, 52], dynamic fuzzer [2] and other IA discovery and VC generation approaches, before presenting an empirical analysis on the detected API misuses and a case study.

5.1 Experiment Setting

Platform. All our experiments were conducted on one 64-bits server running Ubuntu 16.04 with 8 cores (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz), 128GB memory and 3TB hard drive and 2GPUs (12GB Nvidia GPU TiTan X) with CUDA 10.0.

Dataset. To evaluate the effectiveness of Advance, we utilized 5 datasets:

Table 1: Datasets for model training and evaluation.

Libs	D_{iad}			D_{def}		D_{vc}
	S	IA	Non	P_{ia_vp}	P_{ia_np}	P_{ia_cd}
OpenSSL	4686	1305	3381	225	228	163
SQLite	140	102	38	84	123	66
libpcap	100	60	40	36	54	26
libdbus	177	127	50	61	54	57
libxml2	608	521	87	115	336	108
Total	1129	859	270	521	795	420

• *Corpora of library documentations (C_{doc}).* We randomly selected libraries from different categories on the Ubuntu software package website [20], including cryptography (*OpenSSL*), database (*SQLite*), XML file parser (*libxml2*), network packet capture (*libpcap*) and inter-process communication (*libdbus*), and parsed their documents to recover API related information through lxml [16]. Such information is organized in the JSON format *[API name, API parameter, API return type and API description].* In total, we collected the information from 3,581 APIs.

• *Application dataset (D_{app}).* For each selected library, we ran "*apt-cache rdepends*" (a Linux command to manage Linux packages) to search for all the applications integrating the library on Github, Gitlab or sourceforage. In this way, we gathered 39 applications (11 for OpenSSL, 8 for SQLite, 4 for libxml2, 11 for libpcap and 5 for libdbus).

• *Ground-truth API misuse dataset (D_{api}).* To evaluate false negatives incurred by Advance, we collected a set of known API misuses related to the aforementioned five popular libraries as the ground truth. For this purpose, we manually went through CVEs [18] and also checked the commit logs of the applications from Github [13], Gitlab [14], and sourceforge [19], where code patches may be posted, disclosing API misuses. After manually inspecting 6,257 commit logs of 39 applications, we found 66 known API misuses in 27 applications. Among them, 38 misuses and 20 applications are associated with libxml2, libpcap and libdbus. The information of these misuses can be found in Table 5 in Appendix.

• *Ground-truth dataset for IA discovery training and evaluation (D_{iad}).* To generate the training set for S-HAN, we manually annotated 1,305 IAs and 3,881 non-IAs from *OpenSSL*, which is the largest document used in our study. We also performed back-translation [25] to augment the IA dataset using Google translation [15]. Altogether, we collected 2,601 IAs (1,296 IAs from back-translation) and 3,881 non-IAs for model training and cross-validation.

To understand the effectiveness of IA discovery on the testing library documentations, we randomly sampled around 10% of sentences (Column "*S*" in Table 1) from the other 4 library documentations (i.e., SQLite, libxml2, libpcap, libdbus), and annotated them as IA (Column "*IA*" in Table 1) and non-IA (Column "*Non*" in Table 1) for model evaluation.

• *Ground-truth dataset for IA dereference evaluation (D_{def}).* To evaluate IA dereference and VC generation, we randomly sampled around 10% of the IAs reported by the IA discovery step (Section 3.2), which were manually confirmed, to manually inspect their IA dereferences. For this purpose, we annotated 521 IA-VP pairs and 795 IA-NP pairs as the ground truth for IA-API dereference and IA-parameter dereference (Section 3.3) respectively, as shown in Table 1.

Table 2: Effectiveness of individual components

Libs	IA discovery				IA dereference								VC generation		
					API				Parameter						
	ACC	F1	FPR	FNR	ACC	F1	FPR	FNR	ACC	F1	FPR	FNR	Recall	FNR	CD-Cov
OpenSSL	0.91	0.92	0.08	0.09	0.98	0.67	0.02	0	0.83	0.23	0.18	0	0.55	0.45	0.71
SQLite	0.87	0.78	0.11	0.18	0.96	0.67	0.01	0.4	0.92	0.5	0.07	0.29	0.54	0.46	0.59
libpcap	0.84	0.80	0.12	0.23	0.83	0.4	0.15	0.33	0.89	0.5	0.12	0	0.82	0.18	0.77
libdbus	0.86	0.78	0.13	0.16	0.98	0.8	0	0.33	0.85	0.6	0.15	0.14	0.68	0.32	0.79
libxml2	0.94	0.80	0.06	0.09	0.97	0.78	0.01	0.3	0.94	0.73	0.06	0	0.84	0.16	0.91
Average	0.88	0.82	0.1	0.15	0.94	0.66	0.04	0.27	0.89	0.51	0.12	0.09	0.69	0.31	0.75

• *Ground-truth dataset for VC generation evaluation (D_{vc}).* To evaluate the effectiveness of VC generation, we utilized the IAs in D_{def} and manually recovered their associated CDs. These IA-CD pairs form D_{vc} and their number of for each library is shown under "P_{ia_cd}" of Table 1.

5.2 Effectiveness

Here we report the findings made by our evaluation of Advance, first about its overall effectiveness in identifying API misuse from the applications integrating the APIs of libpcap, libdbus and libxml2, and then about the effectiveness of its individual components (i.e., IA discovery, IA dereference and VC generation).

End-to-end effectiveness. Since OpenSSL was used for S-HAN training and later for CD mining along with SQLite, they were excluded in our experiment for fairly evaluating the *end-to-end* effectiveness of Advance. In the experiment, we ran Advance on 20 applications integrating APIs of libpcap, libdbus and libxml2, which reported 92 API misuses. To validate the results, three researchers took 2 days to cross-check D_{api} for known API misuses, and manually verify the results for unknown ones. 83 of the reported API misuses were confirmed (52 undisclosed and 31 disclosed misuses), which yields a precision of 90%. Table 4 in Appendix includes these manually-validated API misuses, along with their security impacts. Note that we also ran Advance on all five library documents as detailed in Section 5.4.

Looking into the 9 false positives, we found that all of them were introduced by CodeQL, which cannot effectively performs a dataflow analysis. Although our verification code were all correctly generated in these cases, CodeQL were found to be less effective in implementing the check, causing the false positives. For example, consider the code snippet in Listing 3 in Appendix: its IA requires the pointer assigned through `xmlGetProp` to be freed, which has been done after the pointer assigned to a list; however, even though the code Advance generates for CodeQL indeed correctly invokes the CodeQL API `TaintTracking` to track this dataflow, the API itself fails to discover the connection between the pointer and the list, thereby falsely claiming discovery of a misuse.

Note that compared with code analysis-based API misuse detectors [30, 52], Advance reports a lower false positive rate (Section 5.3). This is because our approach extracts IAs from API documentation to guide misuse detection in applications, whereas code analysis-based approaches *infer* putative IAs through identifying code invariants, which tend to be less accurate and heavily rely on the quality of D_{app}. Further, Advance is unique in its capability to suppress the false positives incurred by IA discovery and IA dereference, which could be removed by the strict VC generation templates (Section 3.4). As a result, wrongly identified IAs may not be translated into VCs.

To understand false negatives introduced by Advance, we utilized the ground-truth set D_{api} to find out whether it captures known API misuses (related to libxml2, libpcap and libdbus). The study shows that our approach reports 31 out of 38 cases in D_{api} (82%). Among the 7 cases missed, one is introduced by the error in IA dereference, another by the mis-classification of S-HAN, and the all remaining by the failure in CD-to-VCS translation, due to missing CDs (which are rare and therefore are not translated).

Effectiveness of IA discovery. In our study, we first evaluated the effectiveness of IA discovery on 1,305 IAs and 3,881 non-IAs from OpenSSL in D_{iad} using a five-fold cross-validation. Our prototype achieved a false positive rate of 8% and a false negative rate of 9% in finding IA, as shown in Table 2.

Further, using the S-HAN model trained on OpenSSL, we ran IA discovery over the documents of the four other libraries (i.e., SQLite, libpcap, libdbus, libxml2). Altogether, our prototype identified 542, 249, 799 and 1,671 IAs for SQLite, libpcap, libdbus, libxml2, respectively, which yields an average false positive rate of 10% and a false negative rate of 15% on D_{iad} (over the four libraries as shown in Table 1). Table 2 details the experiment results.

When looking into the false positives observed from the model's output, interestingly, we found that most sentences falsely labeled as IAs turn out to indeed contain sentiment terms and state some constraints, which however are supposed to be followed not by the developers who integrate the APIs but by those developing, maintaining or extending the library. For example, the sentence "*Additionally it indicates that the session ticket is in a renewal period and should be replaced*" is falsely labeled as an IA, since it includes the sentiment word "*should*" and describes the required operations to be performed by the OpenSSL library. On the other hand, false negatives apparently were introduced by the sentiment analysis performed by S-HAN, which misses some sentiment terms like "*not safe*", "*is broken*", etc., due to the incompleteness of our training set.

Effectiveness of IA dereference. The effectiveness of the IA dereference was evaluated on D_{vc}. The results are shown in Table 2, where the columns "*API*" and "*Parameter*" present the results for API dereference and parameter dereference respectively. Each kind of dereference was evaluated based upon four metrics *ACC*, *F1*, *FRP* and *FNR*. The API dereference analysis achieves an accuracy of 94%, a F1 of 66%, a false positive rate of 4% and a false negative rate of 27% respectively, which are 89%, 51%, 12% and 9% for the parameter dereference analysis.

For the IA dereference, both FNs and FPs were caused by the incorrect tags generated by the shadow parsing. For example, in the noun phrase (NP) "*file descriptor BIOs*", "*file*" should be labeled as a noun, but the tool we use (StandfordCoreNLP [38]) marks it as a verb, causing the NP to be incorrectly dereferenced. False negatives also occur when the similarity between a VP and the description of its related API is low, due to the limitations of word embedding.

Also, Advance utilized the first sentence of an API's description as the functionality sentence (Section 3.3). To evaluate the effectiveness of this design choice, we sampled 178 API descriptions and found that the first sentences of 157 (88%) are indeed functionality sentences. Also, we manually checked the 21 sentences wrongly labeled. None of them leads to incorrect IA dereference.

Effectiveness of VC generation. The last internal component we evaluated is verification code generation. To evaluate its effectiveness, we checked whether the 420 IA-CD pairs in D_{vc} were generated by our VC generation process, which yields an average recall of 69% and false negative rate of 31% as shown in Table 2.

We further looked into such false negatives in our study, with the following discoveries. Some FNs were introduced by rarely used CDs whose VCSes do not exist in our system. In the other cases, we found that some VCSes could not be handled by CodeQL. For example, consider the statement "*sqlite3_deserialize is only available if SQLite is compiled with the SQLITE_ENABLE_DESERIALIZE option.*"; the CD "*be compiled with*" can be detected by our approach, but cannot be converted to the VCS, since CodeQL cannot check how the application is compiled. As another example, in the IA "*A server application must also call the SSL_CTX_set_tlsext_status_cb function if it wants to be able to provide clients with OCSP Certificate Status responses*", we have no idea whether indeed the developer intends to do so and therefore cannot run CodeQL to check a program's compliance with the IA. Note that VC generation templates used in our study are not narrow, as evidenced by the high coverage of our approach (75% of all the CDs of IAs; see "*CD-Cov*" in Table 2).

Runtime performance. Running Advance on 39 applications associated with 5 libraries (1.47MB files), it took Advance 32.5 hours to finish all the tasks including IA discovery, IA dereference and VC generation. Among the three components, VC generation was the most time-consuming one (31 hours/95%). IA discovery took 1.5 hours (3%) to preprocess data and training S-HAN. It only took 170 seconds to find IAs of 5 libraries. The rest 2% time is used for IA dereference.

5.3 Comparison with the State-of-the-Art

In our research, we firstly compared the end-to-end performance of Advance with that of static API misuse analyzers (e.g., APEx [30] and APISAN [52]) and a dynamic fuzzer (e.g., AFL [2]). Then we further compared the effectiveness of Advance's individual components against their counterparts in the state-of-the-art of other document-based approaches. Note that to the best of our knowledge, there is no end-to-end tool available for detecting API misuse from unstructured library documents.

Comparison with other API misuse detectors. We ran two state-of-the-art static API misuse detection tools (APISAN [52] and APEx [30]) on D_{app}, which reported 150,788 and 1,100 API misuses, respectively. Given the huge number of cases, known high false positive rates of these approaches [30, 52] and the difficulty in validating even a sampled subset (due to the lack of ground truth for the IA semantics inferred), we only cross-checked the results against our ground-truth set D_{api} and the 139 undisclosed but manually-validated API misuses found by Advance. The results are presented under "*APISAN*" and "*APEx*" of Table 4 in Appendix. From the table, we can see that only 15% and 2% API misuses that Advance reports can also be found by APISAN and APEx, respectively. Also, among the 66 misuse cases in D_{api}, APISAN and APEx only find 4 and 2, respectively, whereas Advance detects 54.

Also, when compared with the precision of APISAN and APEx, as reported by the prior work [30, 52] (12% and 21.6%), Advance achieves a much higher precision (90%, see Section 5.2). The reason is that APISAN and APEx rely on program analysis to infer possible IAs (invariants in API uses) for detecting API misuses, which is less reliable and tends to miss legitimate IAs or introduce false ones.

Further, we compared Advance with the most popular dynamic bug detector AFL [2]. In our research, we ran AFL on each application in our dataset (except 1 vulnerable version of ntop, which is too old (16 years ago) to compile). Following Klees et al [33], we set the timeout to 24 hours, and also utilized the default settings to choose initial seeds, i.e., choosing from AFL testcases[4], or the test cases provided by application themselves. From the result shown under "*AFL*" of Table 4, we can see that AFL detects no API misuses. This is because without the guidance of IAs, coverage-based fuzzers like AFL can be hard to trigger the anomalous program behaviors caused by API misuses and in some cases, they fundamentally cannot: for example, those unrelated to memory errors, such as authentication bypass, cannot be found by AFL.

Comparison with other IA discovery approaches. We compared Advance's IA discovery component with two state-of-the-art document-based approaches, one using keywords [47] and the other leveraging grammatical templates based upon shallow parsing (called ALICS) [44] for detecting IAs. In our experiment, we evaluated the approaches on D_{iad} under the settings described in their papers [44, 47]. Figure 5 shows the experiment results in terms of accuracy, F1, FPR and FNR. Our study shows that Advance significantly outperforms both approaches, particularly in terms of accuracy (e.g. 88% vs 44% for ALICS [44]) and F1 score. Also, the false negative rate of Advance is lower.

We also compared our S-HAN with two most popular off-the-shelf classifiers: Text-CNN [32] and RCNN [34]. We trained these models on the training data in D_{iad}. Figure 6 (a) shows the accuracy of the three models (RCNN, Text-CNN, and S-HAN) on D_{iad}. We can see that S-HAN has a much better accuracy, especially when applied to different libraries. This is mainly because the attention mechanism captures the sentiment words used by various developers in different documents.

Comparison with other VC generation approaches. For verification code generation, we compared our approach with three popular tools *tComment* [48], *Toradocu* [27], and *Jdoctor* [22]. Since these tools cannot automatically discover IAs from documents, we utilized the annotated IAs in D_{vc} as their inputs, and evaluated their effectiveness on D_{vc}. Figure 6 (b) illustrates the experiment results, showing that Advance significantly outperforms all these three tools. We observe that Advance incurs a much lower false negative rate than others: these approaches can only handle the IAs

[4]All kinds of file types that are provided by AFL.

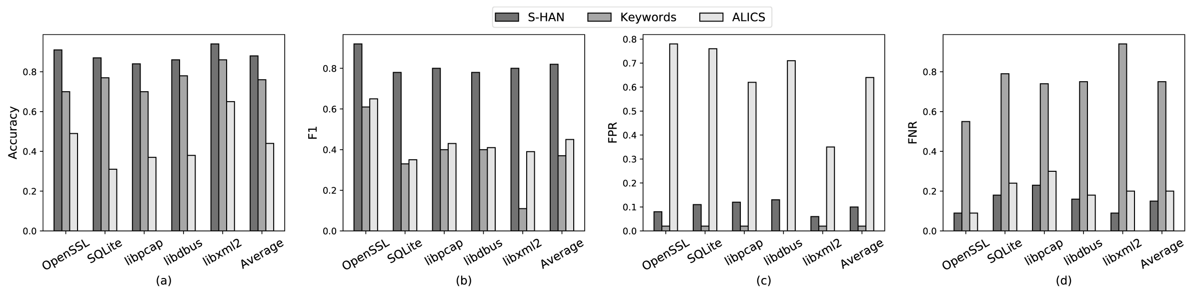


Figure 5: Comparison with other IA discovery approaches (Keywords and ALICS). (a) to (d) represent 4 comparison metrics on 5 evaluated libraries.

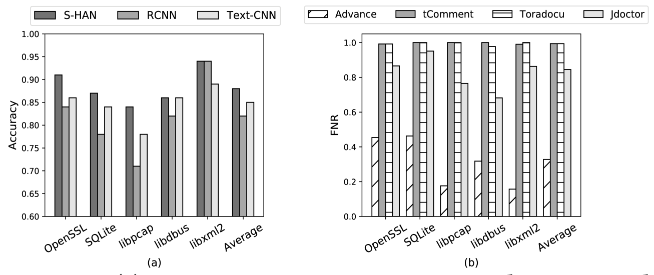


Figure 6: (a) Comparing S-HAN accuracy with RCNN and TextCNN models. (b) Comparing VC generation FNR of Advance with tComment, Teradocu and Jdoctor

with simple arithmetic operations and logic operations, whereas Advance can address more complicated constraints with data flow, control flow and complex context conditions.

5.4 Findings

Running Advance on the documentation of 5 popular libraries (i.e., OpenSSL, SQLite, libpcap, libdbus and libxml2) and 39 applications integrating these libraries, our study uncovered 139 undisclosed and 54 known API misuses (shown in Table 4 in Appendix). Among them, 16 of the undisclosed API misuses have been confirmed by the application developers [5–11]. Also 6 of the known API misuses has been assigned with CVEs. When classifying them using the CWE standard[5], we observe a wide range of vulnerability types: memory corruptions or misuse (e.g., double free (CWE-415), use after free (CWE-416), improper resource shutdown or release (CWE-404)), authentication errors (e.g., improper certificate validation (CWE-295)), incorrect check of return value (CWE-253) and use of obsolete function (CWE-477). Table 4 lists the security impact of 193 API misuses reported by Advance. We can observe serious security implications of these API misuses including information leakage, code execution, system crash, etc. Interestingly, we observed misuse associated with deprecated API, e.g., using the deprecated API `RAND_pseudo_bytes` of OpenSSL will allow remote attackers to defeat the system [12].

Taking a close look at the discovered IAs and the instances of API misuses, we found that around 60% of the violated IAs are post-conditions, such as "*Ownership of the passed parameter tm is not transferred by these functions, so it must be freed up after the call.*" and "*if an error occurs,* `PKCS7_sign_add_signers` *returns NULL*". A hypothesis is that the developer tends to be less careful about the post-conditions once an API has been invoked. Also we found that the IAs of post-conditions or context conditions usually carry more than one CD, due to their relatively complicated grammar structures, while IAs of pre-conditions are usually simple and often used to constrain the input-value range.

[5]CWE defines a list of common software and hardware security weaknesses.

5.5 Case study

Atril [3] is a multi-page document viewer for EPS, DVI, DJVU, XPS, PDF file format. When running Atril 1.24.0 (the latest release), Advance reported an API misuse associated with a NULL dereference bug, which can cause a DOS attack. The bug information from AddressSanitiezer is shown in Appendix Listing 5. The vulnerable function `epub_document_check_hits` is presented in Listing 2.

```
1 guint epub_document_check_hits(...,EvPage *page,...){
2   gchar *filepath = g_filename_from_uri((gchar*)page->
        backend_page,NULL,NULL);
3   htmlDocPtr htmldoc =  xmlParseFile(filepath);
4 + if (!htmldoc) error_handle();
5   htmlNodePtr htmltag = xmlDocGetRootElement(htmldoc);
6 + if (!htmltag) error_handle();
7   htmlNodePtr bodytag = htmltag->xmlChildrenNode;
8   ...
```

Listing 2: An libxml2 API misuse in Atril.

Specifically, `epub_document_check_hits` is a function used to count the number of target strings on a page when searching on the epub format documents. On the line 3, `htmldoc` is assigned with the return value of `xmlParseFile(filepath)`. According to the documentation, `xmlParseFile` returns NULL in some condition (e.g., when `filepath` is non-existent) and thus `htmldoc` needs to be checked in default which, however, has not been followed by Atril. Note that no matter what arguments are passed to `g_filename_from_uri`, `filepath` is NULL, if the file associated with `filepath` does not exist, Further, Atril passes `htmldoc` to `xmlDocGetRootElement` to obtain the root element of the html file. Similar to `xmlParseFile`, `xmlDocGetRootElement` returns NULL when there is no root element (e.g., when `htmldoc` is NULL). Hence, `htmltag` also needs to be checked in default. The violation of these two IAs in the document results in a NULL dereference bug when dereferencing `htmltag` at line 7. To trigger this bug, we set the `filepath` to be an non-existent path. After manually analyzing the source code, we found that `filepath` is a file created by Atril and it does not check whether the file related to `filepath` exists when using. Thus, deleting the file related to `filepath` after being created will trigger the bug. In summary, this bug is caused by the unchecked return value of `xmlParseFile` and `xmlDocGetRootElement`, which is required in the documents in default. It can be triggered through deleting the file `filepath` by another attack process.

Using Advance, we first discovered the IAs "*xmlParseFile returns the resulting document tree if the file was wellformed, NULL otherwise*" and "*xmlDocGetRootElement returns the #xmlNodePtr for the root or NULL*" through S-HAN. Both of them only match the CD "*return*", whose VCS is shown in Listing 4[6] in Appendix. With only one matched CD, that VCS becomes the corresponding VC as the input of CodeQL to discover API misuse. In Listing 4, Line 1 imports the CodeQL modules like Python. The grammars of Line 2-5 are similar to database query language (e.g., MySQL). They query the function invocation `xmlDocGetRootElement` that has not checked the returned `NULL` value and then print the invocation location (i.e., API misuse location).

6 Discussion

With its higher precision than all existing approaches, still Advance introduces false positives and misses some API misuses. These problems mostly come from the limitations of the tools underlying our implementation and unusual IA descriptions present in library documentation. Specifically, CodeQL and the NLP tools (such as StanfordNLP [38]) used in our prototype are imperfect, and their accuracy affects the outcome of our analysis. For example, CodeQL cannot effectively handle complicated data-flow analysis like tracking tainted data across a structure, which leads to a report of false API misuses (Section 5.2). Also, even state-of-the-art NLP techniques cannot effectively handle grammatical errors and ambiguous descriptions, which are widely present in real-world library documentations.

In addition, our approach required an *off-line, one-time* effort to translate popular CDs to VCS (Section 3.4). We acknowledge that our current template-based approach fails to capture the CDs with low frequency, such as "*must be inside an array-typed value*", which only appears once in the library libdbus, or the CD whose IA is hard to comprehend and translate, e.g., "*if it wants to be able to provide clients with OCSP Certificate Status responses*". One possible direction to further automate this step is to customize existing automatic programming techniques [28, 39] to construct the VC for a rare CD based upon its smaller syntactic units such as "array-typed value". Also we found that there is similarity between some rare CDs and popular ones, in terms of their semantics, which could allow us to morph existing VCs for checking these CDs. Such directions will be pursued in our follow-up research.

7 Related Work

Recent years have witnessed numerous studies leveraging text analysis techniques to automatically discover various kinds of bugs, including access control misconfiguration, inappropriate permission request, logic flaws, etc. For example, Zimmeck et al. [55] check the compliance between Android App and privacy requirement. WHYPER [43] and AutoCog [45] investigate whether an Android app properly indicates its permission usage in its app description. Tan et al. [47] extract implicit program rules from comments, then use these rules to automatically detect inconsistencies between comments and source code. Goffi et al. [27] and Blasi et al. [22] generate the test oracle from documentation to dynamically find the inconsistency between documentation and code implementation. Zhong et al. [54] and Pandita et al [42] extract API call sequence information from the documentation to check the inconsistency. Different from previous works, our research provides an end-to-end approach to enable detection of security-critical API misuse from real-world applications.

Considering the approach to discover IAs, previous works mainly utilized the approaches based on keywords [47] or template matching [44]. For instance, Tan et al. [47] utilize a series of pre-defined keywords, such as "should", "must" and so on to extract IAs, which, however, results in a relatively high false-negative rate. Pandita et al. [44] and Chen et al. [23] define shallow parsing templates such as "*(VB) (.)? (PRN)?*" or regex template such as "*Check the seller_id represents the supposed merchant.*", respectively, for IA extraction. In contrast to previous works, in our research, we propose a corpora-insensitive and an efficient IA discovery method based on the bidirectional GRU model with attention.

Another set of studies close to our work is automatic API misuse detection through static or dynamic program analysis. For example, to uncover API specifications, Mithun et al. [21], Kang et al. [30] and Li et al. [36] leverage manually crafted rules to statically find API error-handling blocks (EHBs) which contains the error handling code. Hoan et al. [41] collect the execution paths leading to API calls and then derive potential preconditions for such invocations. Yun et al. [52] generate the symbolic context with relaxed symbolic execution and explore four common API context patterns based on that symbolic context. Maria et al. [31] combine static exception propagation analysis with automatic search-based test case generation to pinpoint crash-prone API misuses in client applications. Considering dynamic analysis for API misuse detection, Wen et al. [50] discover API misuse patterns via mutation analysis. Different from Advance, these methods heavily rely on the code set to infer IAs and utilize manually-crafted rules to capture API misuses, so a low quality code set will cause misuse cases to fall through the cracks.

8 Conclusion

In this paper, we present a new technique to automatically detect API misuses in applications based on analysis of library documentation. Leveraging recent progress in machine learning and NLP, our approach utilizes sentiment analysis to discover the integration assumptions from documentations, tree mining to identify commonly-used CDs and lexical and semantic analysis to resolve implicit references. Running our prototype on the documentations of five libraries and 39 real-world applications integrating these libraries, Advance successfully detected 193 API misuses, with 139 never reported before, outperforming all existing approaches. This study demonstrates that new advancement in intelligent technologies can indeed move security science forward, even on the hard problems long been studied, like API misuse detection.

Acknowledgments

IIE authors are supported in part by Beijing Natural Science Foundation (No.JQ18011), NSFC U1836211, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070), and Beijing Academy of Artificial Intelligence (BAAI).

[6]The CodeQL grammars is shown in https://help.semmle.com/QL/learn-ql/.

References

[1] 2016. stanfordParser. https://nlp.stanford.edu/software/dependencies_manual.pdf. (2016).

[2] 2020. AFL fuzzer. https://lcamtuf.coredump.cx/afl/. (2020).

[3] 2020. Atril for MATE. https://mate-desktop.org/. (2020).

[4] 2020. CodeQL. https://securitylab.github.com/tools/codeql. (2020).

[5] 2020. confirmed bug. https://gitlab.gnome.org/GNOME/anjuta/-/issues/12. (2020).

[6] 2020. confirmed bug. https://gitlab.kitware.com/vtk/vtk/issues/17818. (2020).

[7] 2020. confirmed bug. https://bz.apache.org/bugzilla/show_bug.cgi?id=64264. (2020).

[8] 2020. confirmed bug. https://github.com/hughsie/colord/issues/110. (2020).

[9] 2020. confirmed bug. https://github.com/darktable-org/darktable/issues/6051. (2020).

[10] 2020. confirmed bug. https://github.com/mate-desktop/atril/issues/485. (2020).

[11] 2020. confirmed bug. https://gitlab.gnome.org/GNOME/at-spi2-core/-/issues/24. (2020).

[12] 2020. CVE-2015-8867. https://nvd.nist.gov/vuln/detail/CVE-2015-8867. (2020).

[13] 2020. github. https://github.com/. (2020).

[14] 2020. gitlab. https://about.gitlab.com/. (2020).

[15] 2020. Google translation. https://translate.google.cn. (2020).

[16] 2020. lxml. https://lxml.de/. (2020).

[17] 2020. man3. https://linux.die.net/man/3/. (2020).

[18] 2020. National Vulnerability Datase. https://nvd.nist.gov/vuln/search. (2020).

[19] 2020. sourceforge. https://sourceforge.net/. (2020).

[20] 2020. ubuntu. https://packages.ubuntu.com/en/xenial/libs/. (2020).

[21] Mithun Acharya and Tao Xie. 2009. Mining API error-handling specifications from source code. In International Conference on Fundamental Approaches to Software Engineering. Springer, 370–384.

[22] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 242–253.

[23] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. 2019. Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In 28th {USENIX} Security Symposium ({USENIX} Security 19). 747–764.

[24] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Y. Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. (12 2014).

[25] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding Back-Translation at Scale. (08 2018).

[26] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. AllenNLP: A Deep Semantic Natural Language Processing Platform. arXiv:arXiv:1803.07640

[27] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In Proceedings of the 25th International Symposium on Software Testing and Analysis. 213–224.

[28] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. Foundations and Trends® in Programming Languages 4, 1-2 (2017), 1–119.

[29] huggingface. 2020. neuralcoref. https://github.com/huggingface/neuralcoref. (2020).

[30] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: Automated Inference of Error Specifications for C APIs. In 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). Singapore.

[31] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. 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. 192–203.

[32] Yoon Kim. 2014. Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882 (2014).

[33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2123–2138.

[34] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In Twenty-ninth AAAI conference on artificial intelligence.

[35] Chi Li, Zuxing Gu, Min Zhou, Jiecheng Wu, Jiarui Zhang, and Ming Gu. 2019. API Misuse Detection in C Programs: Practice on SSL APIs. International Journal of Software Engineering and Knowledge Engineering 29, 11&12 (2019), 1761–1779. https://doi.org/10.1142/S0218194019400205

[36] Chi Li, Min Zhou, Zuxing Gu, Ming Gu, and Hongyu Zhang. 2019. Ares: inferring error specifications through static analysis. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 1174–1177.

[37] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 306–315. https://doi.org/10.1145/1081706.1081755

[38] Lynten. 2018. stanfordcorenlp. https://github.com/Lynten/stanford-corenlp. (2018).

[39] Afsaneh Mahanipour and Hossein Nezamabadi-Pour. 2019. GSP: an automatic programming technique with gravitational search algorithm. Applied Intelligence 49, 4 (2019), 1502–1516.

[40] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems. 3111–3119.

[41] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. 2014. Mining preconditions of APIs in large-scale code corpus. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 166–177.

[42] Rahul Pandita, Kunal Taneja, Laurie A. Williams, and Teresa Tung. 2016. ICON: Inferring Temporal Constraints from Natural Language API Descriptions. 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2016), 378–388.

[43] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In 22nd USENIX Security Symposium (USENIX Security 13). USENIX Association, Washington, D.C., 527–542. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/pandita

[44] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In 2012 34th International Conference on Software Engineering (ICSE). IEEE, 815–825.

[45] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 1354–1365.

[46] Fei Sha and Fernando Pereira. 2003. Shallow Parsing with Conditional Random Fields. In Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics. 213–220. https://www.aclweb.org/anthology/N03-1028

[47] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In ACM SIGOPS Operating Systems Review, Vol. 41. ACM, 145–158.

[48] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, 260–269.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In Advances in Neural Information Processing Systems 30, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[50] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing library API misuses via mutation analysis. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 866–877.

[51] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies. 1480–1489.

[52] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 363–378. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

[53] Mohammed Javeed Zaki. 2005. Efficiently mining frequent trees in a forest: Algorithms and applications. IEEE transactions on knowledge and data engineering 17, 8 (2005), 1021–1035.

[54] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, 307–318.

[55] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. 2017. Automated Analysis of Privacy Requirements for Mobile Apps. Korea Society of Internet Information, Korea, Republic of. https://doi.org/10.14722/ndss.2017.23034

[56] Radim Řehůřek. 2019. gensim. https://radimrehurek.com/gensim/. (2019).

9 Appendix

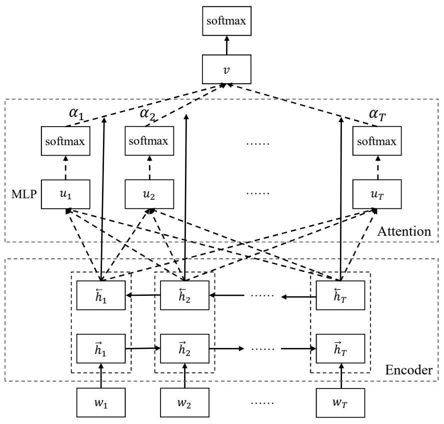


Figure 7: Bidirectional GRU model with Attention

```
Algorithm 1 Traverse the CD tree
Input: $CD\_tree$
Output: $VC$
1: while $get\_nodes(CD\_tree) > 2$ do
2:     $leaf \leftarrow select\_leaf(CD\_tree)$
3:     $siblings \leftarrow get\_siblings(leaf)$
4:     $parent \leftarrow get\_parent(leaf)$
5:     $new\_node \leftarrow gen\_node(siblings, leaf, parent)$
6:     $replace(CD\_tree, parent, new\_node)$
7: end while
8: $VC \leftarrow CD\_tree.root()$
```

Table 3: Dependent tools and SLoC of each component in Advance

Components	Dependent tools	SLoC
IA discovery	Gensim, Stanford-NLP, Keras	3K
IA derefence	Gensim, Stanford-NLP, NLTK	1K
VC generation	Allennlp, CodeQL	1K

```
ptr = xmlGetProp (...);
list = g_list_append (list, ptr);
for (iter = g_list_first (list); iter != NULL; iter =
     g_list_next (iter))
  g_free (iter->data);
```

Listing 3: An example of false positives

```
import cpp, nullCheck
from FunctionCall fc
where fc.getTarget().hasName("xmlDocGetRootElement")
      and not nullcheck(fc)
select fc.getLocation()
```

Listing 4: Verification code

```
==167813==ERROR: AddressSanitizer: SEGV on unknown address 0
     x000000000018 (pc 0x7f92a84b1b31 bp 0x7ffdba993ba0 sp 0
     x7ffdba993b50 T0)
==167813==The signal is caused by a READ memory access.
==167813==Hint: address points to the zero page.
 #0 0x7f92a84b1b30 in epub_document_check_hits /atril/backend/epub
      /epub-document.c:186
 #1 0x7f92fa1418a6 in ev_document_find_check_for_hits /atril/
      libdocument/ev-document-find.c:59
 #2 0x7f92fa085698 in ev_job_find_run /atril/libview/ev-jobs.c
      :1378
 #3 0x7f92fa07f7ef in ev_job_run /atril/libview/ev-jobs.c:192
 #4 0x7f92fa087a86 in ev_job_idle /atril/libview/ev-job-scheduler.
      c:199
 #5 0x7f92f92e9dfd in g_main_context_dispatch (/lib/x86_64-linux-
      gnu/libglib-2.0.so.0+0x4fdfd)
 #6 0x7f92f92ea1af  (/lib/x86_64-linux-gnu/libglib-2.0.so.0+0
      x501af)
 #7 0x7f92f92ea23e in g_main_context_iteration (/lib/x86_64-linux-
      gnu/libglib-2.0.so.0+0x5023e)
 #8 0x7f92f94f624c in g_application_run (/lib/x86_64-linux-gnu/
      libgio-2.0.so.0+0xd924c)
 #9 0x556d3653a419 in main /atril/shell/main.c:287
 #10 0x7f92f8f9abba in __libc_start_main ../csu/libc-start.c:308
 #11 0x556d364e1349 in _start (/usr/local/bin/atril+0x3a349)
SUMMARY: AddressSanitizer: SEGV /atril/backend/epub/epub-document.
     c:186 in epub_document_check_hits
==167813==ABORTING
```

Listing 5: Sanitizer information of the case study.

Table 4: List of manually-validated API misuses reported by Advance, including 139 undisclosed (labeled with "*" in the Column "*Advance*") and 54 disclosed API misuses.

Lib	App	SLoC	API Misuse	Impact	Advance	APISAN	APEx	AFL
libdbus	afterstep-devel	240 K	missing `dbus_free()` after `dbus_malloc()`	DoS	1*	0	0	0
	at-spi2-core	22 K	missing `dbus_free()` after `dbus_message_iter_get_signature()`	DoS	1*	0	0	0
					5	0	0	0
	avahi	44 K	deprecated `dbus_message_iter_get_array_len()`	malfunction	1	0	0	0
	BlueZ	191 K	missing `dbus_message_get_sender()` check	system crash	1	0	0	0
libpcap	arp-scan	5 K	incorrect `pcap_dispatch()` check	malfunction	1*	0	0	0
			incorrect `pcap_set_timeout()` argument	malfunction	1	0	0	0
	arping	2 K	incorrect `pcap_dispatch()` check	malfunction	1	0	0	0
	ettercap	75 K	deprecated pcap_lookupdev	malfunction	5	0	0	0
			incorrect `pcap_dump()` argument	malfunction	2	0	0	0
			incorrect `pcap_setfilter()` check	malfunction	1*	0	0	0
			missing `pcap_freealldevs()` after `pcap_findalldevs()`	DoS	1*	0	0	0
			missing `pcap_lookupdev()` check	system crash	1	0	0	
	freeradius	147 K	incorrect `pcap_open_live()` argument	malfunction	1*	0	0	0
			`pcap_create()` `pcap_activate()` not available	malfunction	1	0	0	0
	knock	2 K	incorrect `pcap_open_live()` argument	malfunction	1	0	0	0
	libnet	24 K	missing `pcap_freealldevs()` after `pcap_findalldevs()`	DoS	1	0	0	0
	ntop	6 M	missing `pcap_freealldevs()` after `pcap_findalldevs()`	DoS	1	0	0	
	tcpdump	96 K	missing `pcap_freealldevs()` after `pcap_findalldevs()`	DoS	2	0	0	0
			using `pcap_geterr()` return value after `pcap_close()`	system crash	1	0	0	0
	tcpreplay	62 K	incorrect `pcap_open_live()` argument	malfunction	1*	0	0	0
	wireshark	5 M	incorrect `pcap_list_datalinks()` check	system crash	1	0	0	0
			incorrect `pcap_open_live()` argument	malfunction	2*	0	0	0
			incorrect `pcap_set_tstamp_type()` check	malfunction	1*	0	0	0
libxml2	abiword	541 K	missing `xmlFree()` after `xmlGetProp()`	DoS	2	0	0	0
			missing `xmlFree()` after `xmlNodeGetContent()`	DoS	4	0	0	0
	anjuta	12 M	missing `xmlDocGetRootElement()` check	system crash	5*	0	0	0
			missing `xmlFree()` after `xmlGetProp()`	DoS	9*	0	0	0
			missing `xmlFree()` after `xmlNodeGetContent()`	DoS	2*	0	0	0
			missing `xmlNodeGetContent()` check	system crash	2*	0	0	0
	atril	94 K	missing `xmlDocGetRootElement()` check	system crash	5*	0	1	0
			missing `xmlFree()` after `xmlNodeGetContent()`	DoS	4*	0	0	0
	vtk	4 M	missing `xmlDocGetRootElement()` check	system crash	1*	0	0	0
			missing `xmlFree()` after `xmlGetNsProp()`	malfunction	2*	0	0	0
			missing `xmlFree()` after `xmlGetProp()`	DoS	8*	0	0	0
			missing `xmlFree()` after `xmlNodeGetContent()`	DoS	4*	3	0	0
OpenSSL	dovecot	401 K	missing `BIO_reset()` check	malfunction	1	0	0	0
			missing `EVP_PKEY_CTX_new()` check	malfunction	1	1	1	0
	httpd	422 K	missing `X509_free()` after `SSL_get_peer_certificate()`	Privacy leakage	2*	2	0	0
	mutt	113 K	`X509_get_notBefore()` and `X509_get_notAfter()` derepcated	malfunction	6	0	0	0
	ntp	214 K	incorrect `EVP_Verifyinal()` check	authentication bypass	1	0	0	0
			incorrect `RSA_private_derypt()` argument	code execution	1	0	0	0
	openfortivpn	6 K	incorrect `X509_check_host()` check	authentication bypass	1	0	0	0
	openvpn	90 K	freeing `SSL_get_certificate()` return value	system crash	1	0	0	0
			incorrect `ASN1_STRING_to_UTF8()` check	DoS	3	0	0	0
	ovs	483 K	missing `X509_free()` after `SSL_get_peer_certificate()`	DoS	1	1	0	0
	PHP	1 M	deprecated `RAND_pseudo_bytes()`	cryptographic issues	1	0	0	0
	SPICE	190 K	incorrect `RSA_private_decrypt()` argument	code execution	1	0	1	0
	unbound	88 K	missing `EVP_PKEY_assign_RSA()` check	malfunction	1	0	0	0
SQLite	anope	332 K	missing `sqlite3_close()` after `sqlite3_open_v2()`	Dos	1	0	0	0
			missing `sqlite3_finalize()` after `sqlite3_open_v2()`	DoS	1	1	0	0
	bibledit-gtk-old	110 K	missing `sqlite3_close()` after `sqlite3_open()`	information leakage	3*	2	0	0
			missing `sqlite3_free()` after `sqlite3_exec()`	Dos/malfunction	45*	17	0	0
	cmtk	334 K	missing `sqlite3_finalize()` after `sqlite3_prepare_v2()`	information leakage	2*	0	0	0
	colord	110 K	missing `sqlite3_free()` after `sqlite3_exec()`	Dos	2*	0	0	0
	darktable	385 K	calling `sqlite3_config()` after `sqlite3_initialize()`	malfunction	1	0	0	0
			missing `sqlite3_finalize()` after `sqlite3_prepare_v2()`	information leakage	8*	0	0	0
			missing `sqlite3_free()` after `sqlite3_exec()`	Dos	1*	0	0	0
	librdf	2 M	missing sqlite3_free after `sqlite3_exec()`	DoS	1	1	0	0
	libspatialite	616 K	missing `sqlite3_free_table()` after `sqlite3_get_table()`	DoS	24*	0	0	0
All	/	36 M	/	/	193	28	3	0

Table 5: Known API misuse dataset (D_{api})

Applications	Libraries	Version	Misuses	IA
at-spi2-core	libdbus	925201d	5	The returned string must be freed with dbus_free().
avahi	libdbus	28eb71a	1	This function is deprecated on the grounds that it is stupid.
BlueZ	libdbus	d3ae2d6	1	dbus_message_get_sender returns the unique name of the sender or NULL
arp-scan	libpcap	f013b45	1	We recommend always setting the timeout to a non-zero value unless immediate mode is set, in which case the timeout has no effect.
arping	libpcap	b37fb24	1	pcap_dispatch() returns the number of packets processed on success; this can be 0 if no packets were read from a live capture or if no more packets are available in a savefile. It returns PCAP_ERROR if an error occurs or PCAP_ERROR_BREAK if the loop terminated due to a call to pcap_breakloop() before any packets were processed.
ettercap	libpcap	89b5542	5	This interface is obsoleted by pcap_findalldevs.
ettercap	libpcap	dfcabfc	2	If called directly, the user parameter is of type pcap_dumper_t as returned by pcap_dump_open().
ettercap	libpcap	891a281	1	If there is an error, or if pcap_init() has been called, NULL is returned and errbuf is filled in with an appropriate error message.
freeradius	libpcap	57fbb95	1	pcap_create() and pcap_activate() were not available in versions of libpcap prior to 1.0
knock	libpcap	4b8ad4d	1	you should use a non-zero timeout
libnet	libpcap	008c994	1	The list of devices must be freed with pcap_freealldevs(), which frees the list pointed to by alldevs.
ntop	libpcap	66f6f48	1	The list of devices must be freed with pcap_freealldevs(), which frees the list pointed to by alldevs.
tcpdump	libpcap	39be365	1	you must use or copy the string before closing the pcap_t.
tcpdump	libpcap	224b073	2	The list of devices must be freed with pcap_freealldevs(), which frees the list pointed to by alldevs.
wireshark	libpcap	51a99ca	1	pcap_list_datalinks() returns the number of link-layer header types in the array on success, PCAP_ERROR_NOT_ACTIVATED if called on a capture handle that has been created but not activated, and PCAP_ERROR on other errors.
anope	SQLite	2a5e782	1	resources associated with the database connection handle should be released by passing it to sqlite3_close() when it is no longer required.
anope	SQLite	aeefe16	1	The calling procedure is responsible for deleting the compiled SQL statement using sqlite3_finalize() after it has finished with it
darktable	SQLite	70820b1	1	The sqlite3_config() interface may only be invoked prior to library initialization using sqlite3_initialize() or after shutdown by sqlite3_shutdown().
librdf	SQLite	5d074c1	1	To avoid memory leaks, the application should invoke sqlite3_free() on error message strings returned through the 5th parameter of sqlite3_exec() after the error message string is no longer needed.
abiword	libxml2	80fee4c	2	It's up to the caller to free the memory with xmlFree().
abiword	libxml2	ebcc445	4	It's up to the caller to free the memory with xmlFree().
dovecot	OpenSSL	0eaf77d	1	EVP_PKEY_CTX_new(), EVP_PKEY_CTX_new_id(), EVP_PKEY_CTX_dup() returns either the newly allocated EVP_PKEY_CTX structure or NULL if an error occurred.
dovecot	OpenSSL	394391e	1	BIO_reset() normally returns 1 for success and 0 or -1 for failure.
mutt	OpenSSL	101e05d6	6	X509_get_notBefore() and X509_get_notAfter() were deprecated in OpenSSL 1.1.0
ntp	OpenSSL	2383333	1	to must point to RSA_size(rsa) bytes of memory.
ntp	OpenSSL	c70fc4b	1	EVP_VerifyFinal() returns 1 for a correct signature, 0 for failure and -1 if some other error occurred.
openfortivpn	OpenSSL	07946c1	1	The functions return 1 for a successful match, 0 for a failed match and -1 for an internal error: typically a memory allocation failure or an ASN.1 decoding error.
openvpn	OpenSSL	f755c99	1	They returned internal pointers that must not be freed by the application program.
openvpn	OpenSSL	0007b2d	3	The length of ASN1_STRING_to_UTF8_APIParam_1 is returned or a negative error code
ovs	OpenSSL	9da8b2f	1	The X509 object must be explicitly freed using X509_free().
PHP	OpenSSL	7a4584d	1	RAND_pseudo_bytes() was deprecated in OpenSSL 1.1.0
SPICE	OpenSSL	ef9a8bf	1	to must point to RSA_size(rsa) bytes of memory.
unbound	OpenSSL	ffed368	1	EVP_PKEY_assign_RSA(), EVP_PKEY_assign_DSA(), EVP_PKEY_assign_DH(), EVP_PKEY_assign_EC_KEY(), EVP_PKEY_assign_POLY1305() and EVP_PKEY_assign_SIPHASH() return 1 for success and 0 for failure.
Total	/	/	66	/