Li *et al. Cybersecurity* (2022) 5:14
https://doi.org/10.1186/s42400-022-00118-9

Cybersecurity

RESEARCH Open Access

The inconsistency of documentation: a study of online C standard library documents

Ruishi Li[1,2], Yunfei Yang[1,2], Jinghua Liu[1,2], Peiwei Hu[1,2] and Guozhu Meng[1,2*]

Abstract

The C standard libraries are basic function libraries standardized by the C language. Programmers usually refer to their API documentation provided by third-party websites. Unfortunately, these documents are not necessarily complete or accurate, especially for constraint sentences of API usage, which are called Security Specifications (SSs). SS issues can prevent programmers from following obligatory constraints, which results in API misuse vulnerabilities. Previous work studying SS issues could only find certain types of inaccurate SSs through checking the compliance between API usage and existing SSs. Therefore, we propose a novel approach SSeeker for quickly discovering missing and inaccurate SSs through the inconsistency of semantically similar SSs. More specifically, SSeeker first completes broken sentences and discovers SSs from them by judging their constraint sentiment. Then SSeeker puts semantically similar SSs from different sources into a group, which can be used to discover missing or inaccurate SSs. With the help of SSeeker, we investigated 4 popular online third-party C standard library documents, studied their conformity with the C99 standard, analyzed their APIs and SSs, and discovered 92 prototype issues, 15 web page issues, and 96 SS issues.

Keywords: Documentation issues, Security specification, Standard library

Introduction

Software libraries provide Application Program Interfaces (APIs) for users to implement specific functionality when developing programs. While using these APIs, programmers are expected to follow certain constraints on their inputs (e.g., the size limit on argument), outputs (e.g., check if the return value is NULL), and invocation sequences (e.g., call another API before invocation). These constraints are called Security Specifications (SSs) and are described in the API documents along with these APIs' prototypes and functionality descriptions. If these SSs in the API documents are not followed, it may introduce API misuse bugs and cause severe security problems (Liu et al. 2020; Yu et al. 2021), e.g., execution of arbitrary code (CVE-2005-3346) in "SS and API misuse" section. Not only do programmers trust the reliability of the API documents but also some API misuse detection studies (Tan et al. 2007, 2012; Blasi et al. 2018; Lv et al. 2020) depend on the documents. However, according to our observation, even popular API document has some documentation issues, such as wrong prototype and inaccurate or lacking SS. For example, the cplusplus document (cplusplus 2021a) of `nanl` has a prototype issue. It says the return type is `float`, which is wrong and should be `long double` according to the official C99 standard document (ISO 2021a). The SSs of APIs could also be inaccurate. An SS of `memcpy` API in Microsoft documentation (Microsoft 2021a), "*Make sure that the destination buffer is the same size or larger than the source buffer.*", states the size limit on arguments. However, when the "*destination buffer*" is larger than the number characters to copy ("*count*"), it doesn't matter even if the "*destination buffer*" is smaller than the "*source buffer*", which is also very common during programming. Besides, missing SS is also very common, e.g., we found 95 missing SSs of 4 popular document websites shown in "Discoveries" section.

*Correspondence: mengguozhu@iie.ac.cn
[1] SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
Full list of author information is available at the end of the article

Previous research (Tan et al. 2007, 2012; Blasi et al. 2018) to discover SS issues mainly detect the inconsistency between API usage code and SSs of API document or library code comment. For example, `in2000_bus_reset` calls `UnlockSet` before `ConvertToSID`, which is inconsistent with the comment of `ConvertToSID` ("*Caller must hold cache lock ...*") in Fig. 1. After manual review, library maintainers confirmed this SS is wrong. However, due to the designs of their approaches and the limitations of code analysis used, these work could not check some SSs on API usage, e.g., SSs about library compilation options. Besides, they can only find the inaccuracy of SSs only if SSs are inconsistent with API usage but cannot find the absence of SSs. Compared with previous work, our study can not only uncover a wider range of SS issues but also uncover missing SSs because we study inconsistencies across documents rather than documentation versus code.

Challenges. Sentences in API documents are often loosely organized and broken, i.e., lacking constituents and having coreferences, which will reduce the performance of the following text parsing. For example, this sentence "*Copies the first num characters of source to destination*" lacks the subject constituent. Except the coreference of pronouns (e.g., "*it*", "*they*"), API documents have specific coreference of API and arguments, such as "*The function*" in "*The function shall return no value and take no arguments*". The lacking constituent and referent can be the API of this sentence or its arguments, return value, or even other APIs that appears around this sentence. Another challenge is to group SSs with similar semantic when the group number is unknown and the SSs number is small. According to our discoveries in "APIs and SSs of the Documentation" section most of the API documents have 0 to 4 SSs so the SSs to be grouped would be only dozens. Besides, the quantity of constraint types is different among APIs and sealed before grouping.

Therefore, we proposed a new approach SSeeker to find missing or inaccurate SSs through the inconsistency of semantically similar SSs. It first completes broken sentences using context-sensitive dependency parsing and Part-of-speech (POS) tagging. It then discovers SSs from sentences using sentiment analysis since SSs often have strong sentiments to restrict what API users should do when using these APIs. Next, for every API, SSeeker collects its SSs from different document sources and generates their semantically meaningful sentence embeddings. Last, SSeeker uses a greedy algorithm to group these embeddings into an indefinite number of SS groups. It would be easy to find missing SS if one document source lacks SS in one SS group and wrong SS if this SS is different from other SSs in the group.

To evaluate the effectiveness of SSeeker, we chose 4 popular third-party websites among programmers including cplusplus (cplusplus 2021b) (we call it "cpp" for short in our paper), cppreference (cppreference 2021) ("cppref" for short), Linux manual page (man7 2021a) ("Linux" for short), and Microsoft documentation (Microsoft 2021a) ("Microsoft" for short). 10% of sentences sampled from these documents were used for evaluation of SS discovery, which was implemented based on a pre-trained Sentence-Hierarchical Attention Network (S-HAN) (Lv et al. 2020). It achieved 91% accuracy in the evaluation ("Experiments for Answering RQ1" section), which performs much better than the keyword-based method (Tan et al. 2007). In order to measure the performance of sentence embedding generation, we compared our utilized SBERT (Sentence-BERT) (Reimers and Gurevych 2019) with four models based on Word2Vec and the SBERT model outperforms others ("Experiments for Answering RQ2" section).

With the help of SSeeker, we further investigated these 4 online C standard library API documents for C99, since C99 is widely used for C language programming. Note that our approach can be directly applied to other standards, e.g. C11. We revealed the correlation between their APIs and SSs, discovered 3 types of API documentation issues and provided advice for document maintainers.

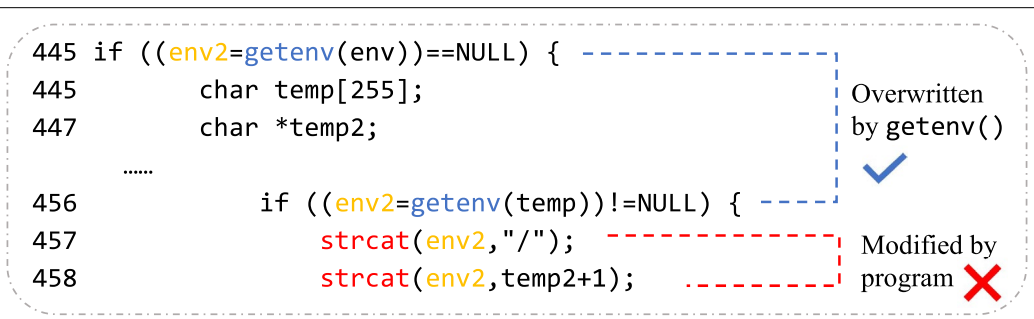


Fig. 1 An example of the API misuse

The contributions of this paper are summarized as follows:

- Proposed an approach *SSeeker* to quickly discover missing or inaccurate SS through sentiment analysis, and semantic similarity analysis.
- Analyzed the C standard library documents of 4 popular third-party websites, and found 92 prototype issues, 15 web page issues, and 96 SS issues.
- Categorized the documentation issues on third-party websites and provided suggestions for documentation maintainers to write secure API documentation.

Paper structure. The rest of the paper is structured as follows. "Background" section describes the related work, while "Approach" section illustrates the design of our proposed approach SSeeker. We present our evaluation of SSeeker in "Evaluation" section and show the discovery when investigating the online documents with the help of SSeeker in "Discoveries" section. "Discussion" section presents a discussion of our work and "Related work" section concludes.

Background

SS and API misuse

API documents of software library not only declare the basic functionality of provided functions, but also indicate some constraints, i.e., *security specifications (SSs)*, that developers need to comply with when using this API, e.g., value range for arguments, a need to check the return value, and API call sequence. Otherwise, it may cause severe security issues (such as buffer overflow, privilege escalation, use-after-free, and etc.). For example, there is an SS in the `getenv` function of the C99 standard: *"The getenv function returns a pointer to a string ... The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the getenv function."* ISO (2021a), which restricts developers from modifying the return value except using `getenv`. Figure 1 shows a code snippet using `getenv` in OSH 1.7-14. OSH didn't follow this SS but modified the return value of `getenv` directly thus introducing a buffer overflow vulnerability, which allows execution of arbitrary code (CVE-2005-3346) (NVD 2021).

C standard library

The C standard library (Wikipedia 2021a) is a basic function library for the standardized C programming language and provides users with unified APIs, which ensures the platform portability of software written in the C language. The library is continuously improved with the revision of the C language standards (Wikipedia 2021b). Named by the released year, there are C89 (Wikipedia 2021c), C90 (Wikipedia 2021d) (same with C89), C95 (Wikipedia 2021e), C99 (Wikipedia 2021f), C11 (Wikipedia 2021g), and C17 (ISO 2021b) (also known as C18) standards, among which C99 is one of the most widely used one. The C99 standard library contains 24 header files and 463 functions (not considering macros). The official documentation of C standard library is within the hundreds of pages PDF file specified by the International Organization for Standardization (ISO) and it is not free. Instead of buying official PDFs, programmers usually read API reference documents on third-party websites, e.g., cplusplus (cplusplus 2021b) and cppreference (cppreference 2021). Unlike tidy official standard documents released after years of discussion, third-party documents are more frequently updated by developers with what they think is important for users. This makes third-party documents contain more SSs than the official documents.

Natural language processing

Here is a brief introduction of a set of NLP techniques we leveraged in our research.

Dependency parsing

Dependency parsing is the process of analyzing the syntactic structure in a sentence and extracting grammatical relations between terms. Its result is this sentence's dependency tree, where the root is the verb of a clause and other words are linked to the root by relations. Every relation has one headword and a dependent that modifies the head and point to it by a directed line in the tree. For example, Fig. 3a shows the dependency tree of one sentence. The verb "*Invokes*" is the root and "*processor*" is the object through "obj" relation. In our research, we utilize the Stanford parser (StanfordParser 2016) to detect if a sentence lacks constituents and resolve specific coreferences.

Part-of-speech (POS) tagging

POS tagging assigns POS labels, e.g., noun, adjective, and verb, to words of a sentence. POS tag of one word is decided by its definition and context since the same word can have more than one POS at different times. For example, the POS of word "*command*" is a noun (tagging is "*NN*") as shown in Fig. 3a. However, this word could also be a verb in a different context. Stanford POS Tagger (Group 2021a) is one of the state-of-the-art POS tagging tools so we utilize it to help sentence completion.

Coreference resolution

Coreference occurs when some expressions refer to the same entity in a text and coreference resolution is the process of finding them. For every coreference, there

would be a referent, which is usually a full form, and one expression refers to it. This expression would often be an abbreviated form for traditional coreference, e.g., pronouns "*it*" and "*they*". In our research, API documents may have specific coreferences, e.g., "*the function*" or "*the return value*", referring to APIs, their arguments, or return values. In our research, we leverage Stanford Deterministic Coreference Resolution System (Group 2021b) to resolve traditional coreferences.

Sentiment analysis

Sentiment analysis, also known as opinion mining, is the process of identifying and extracting the opinion and subjective information in a text. It can be used as a text classification tool to judge the underlying sentiment is positive, negative, or neutral. In recent years, more Deep Learning-based classifiers were proposed for sentiment analysis, e.g., Text-CNN (Kim 2014), RCNN (Lai et al. 2015), and HAN (Yang et al. 2016). Since HAN outperforms the other two approaches in previous research (Lv et al. 2020), we employ the S-HAN model (Lv et al. 2020) modified based on HAN.

Approach

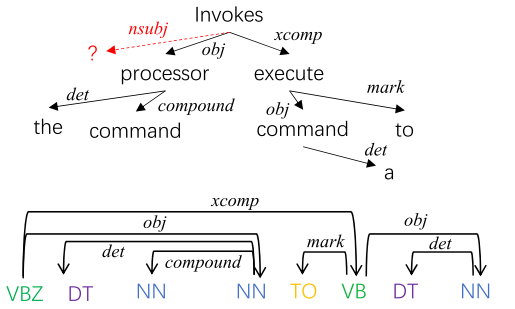
In this section, we elaborate on the design of SSeeker, which can help quickly discover the missing or inaccurate security specifications. We first give an overview of the design and then describe the individual components.

Overview

Architecture. Figure 2 illustrates the architecture of our approach SSeeker, including four components: sentence completion, SS discovery, sentence embedding generation, and SS grouping. SSeeker takes API documents from different sources as input. During the sentence completion step, SSeeker utilizes dependency parsing to decide if one sentence lacks constituents and completes the sentence with the consideration of its POS and its context. Then SSeeker uses a sentiment analysis model S-HAN (Lv et al. 2020) to judge if one sentence is SS by detecting the specific emotional tone of SS. Next, SSeeker collects SSs from different document sources for every API and generates their sentence embeddings using SBERT. Last, SSs of every API would be grouped into several groups by a greedy algorithm according to their semantic similarity.

Sentence completion

Due to the casual writing style of API documentation maintainers, the documents of C language-based libraries are loosely organized and often lack constituents or have coreferences as shown in the "Challenges" of "Introduction" section. These flaws of sentences would impede the following sentiment analysis and semantic grouping due to the missing constituents or unsolved referents.



(a) Dependency tree and POS tags of the example sentence

(Subject) VB / VBZ / VBD / VBG / VBP
(system_API) invokes the command processor to execute a command

(Subject is) NN / NNS / NNP / NNPS
(strstr_param_1 is) C string to be scanned.

(Subject is) JJ / JJR / JJS / ADJP
(fdim_API is) equivalent to fmax(x-y, 0) except for the NaN handling requirements.

(b) Rules for constituent completion

Fig. 3 Constituent completion

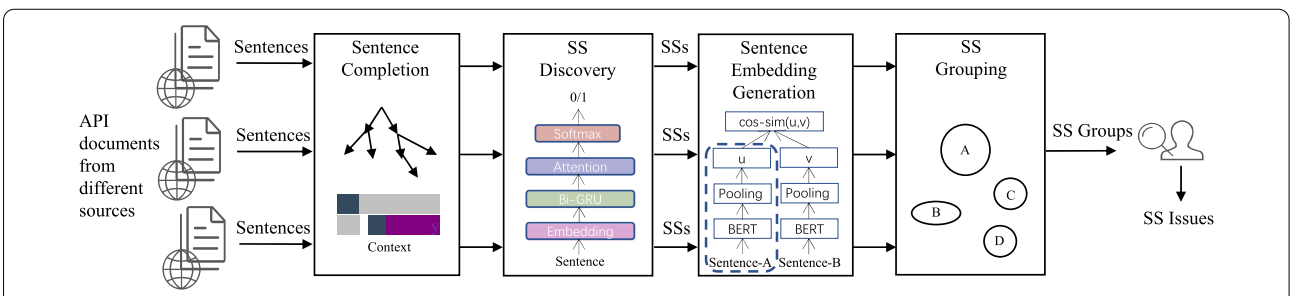


Fig. 2 The design of our approach

Therefore, SSeeker utilizes dependency parsing and POS tagging to complete the lacking constituents and resolve coreferences with the consideration of context.

Constituent completion

Specifically, we utilize the dependency parsing technique to generate the dependency tree of one sentence and check if this tree misses basic grammatical relations, e.g., "nsubj" and "obj". For example, Fig. 3a shows the basic dependency tree of "*Invokes the command processor to execute a command*" in the above and another one with POS. Examining this tree, we could easily find that the root word lacks the "nsubj" relation, which means this sentence lacks the subject constituent. Next, SSeeker infers the lacking subject according to the context of this sentence. Based on our observation, it can be decided by which paragraph this sentence belongs to, i.e., the paragraph of API or different arguments or the return value. For this sentence, it lies in the paragraph of `system` so the subject word should be `system`. Then SSeeker generates the POS of this sentence and decides how to add the dependent word to this sentence according to the POS of headword and our summarized rules. For this example, the head and dependent words of missing "nsubj" relation are "*invokes*" and `system` respectively and the "*invokes*" is recognized as a 3rd person sigular present verb (tagged as "*VBZ*") so the word `system` should be added to the beginning of this sentence. We have summarized three rules for adding the subject and elaborated them with examples in Fig. 3b. These three rules correspond to the cases when the root word is a verb, noun, and adjective.

Coreference resolution

For the traditional coreference of pronouns, we utilize AllenNLP (Gardner et al. 2017) tool to resolve them. For the specific coreference, e.g., "*the function*" and "*the return value*", we infer the referent according the the context of this sentence. Like in the "Constituent Completion", the referent is decided by the paragraph this sentence lies in. Specifically, the referent would be the subject of the previous sentence if this subject is another API, its arguments, or its return value.

SS discovery

Because of the loose structure of API documents and the different writing styles of library maintainers, the form of SSs varies greatly. Previous work (Tan et al. 2012, 2007) using keyword-based or rule-based method can only detect limited SSs and have high false negative rate (86% in "Experiments for answering RQ1" section). Despite the varying syntax structure, SSs have a specific emotional tone, which shows the developers the explicit or implicit directions constraining what developers should follow. For example, "*The strings may not overlap, and the destination string dest must be large enough to receive the copy*" in `strcpy` from Linux man page has very strong sentiment ("*must*" and "*may not*") to stress the constraints; "*The behavior is undefined if either dest or src is a null pointer.*" in `memcpy` from cppreference also implies the constraint on arguments. Therefore, we choose a sentiment analysis-based boolean classifier to detect SSs.

The classifier is composed of a Bi-GRU-based encoder (Chung et al. 2014) and an attention mechanism (Vaswani et al. 2017), as shown in the second step of Fig. 2. The first layer is an embedding layer, which generates the word vectors w_i of one sentence using the pre-trained Word2Vec (Mikolov et al. 2013) model. Then w_i are fed into the Bidirectional GRUs (Bi-GRU) to learn the context of this sentence by collecting information from both directions. After that, the word annotation vectors h_i outputted by Bi-GRU would be input into an attention layer, which first generates u_i using the Multilayer Perceptron (MLP) and then the attention weight α_i using the softmax function, as in Eq. 1, where u_w is a word-level context vector. Last, the sentence vector v is produced by summing the word annotation vectors h_i with its attention weights α_i and ready to be inputted into the softmax function to give boolean result whether this sentence is an SS.

$$\alpha_i = \frac{\exp\left(u_i^T u_w\right)}{\sum_i \exp\left(u_i^T u_w\right)} \tag{1}$$

We choose the pre-trained model S-HAN, which was trained on a dataset collected from OpenSSL documentation and annotated manually. The dataset consists of 2,601 SSs (1,296 SSs from back-translation) and 3,881 non-SSs. S-HAN achieved an accuracy 91% for Standard C library documentation classification task in our evaluation of effectiveness ("Effectiveness" section), higher than 88% in the original work.

Sentence embedding generation

Online documents from different sources should provide similar SSs for the same API. Based on our observation, SSs with the same meaning may not have the same grammatical structure but similar semantics. For example, programmer should not modify the return value of the `char *getenv( const char *name )`. This SS is described as "*The caller must take care not to modify this getenv_Param0, ...*" in Linux and "*Modifying the getenv_Param0 returned by getenv_API invokes undefined behavior.*" in cppref, where the "*getenv_Param0*" means the return value of `getenv`. The variety of syntax and words makes it hard for knowledge-based approaches to conduct the sentence similarity comparison, so SSeeker

utilizes a model to generate semantically meaningful sentence embeddings.

Specially, we choose SBERT (Sentence-BERT) model, which modifies the pretrained BERT model by using siamese and triplet network structures so that it can generate sentence embeddings suitable for semantic comparison. As shown in Fig. 2, a pooling operation is added to the output of BERT so that SBERT can derive fixed-sized sentence embedding vectors, which can be compared with cosine-similarity.

SS grouping

Different documents of one API are supposed to provide semantically similar SSs, which can be divided into different SS groups according to their meaning. The number of SS groups differs among APIs, especially when the problem of lacking SSs can happen as discussed in "Introduction" section. In addition, the SSs number of most documents ranges between 0 and 4 as shown in our discoveries "APIs and SSs of the documentation" section. Even if we collect documents from four sources for one API, the SSs to be grouped would be no more than 20. In order to group a small number of SSs whose group number is unknown, we propose a greedy algorithm based on the semantic similarity of sentence embeddings.

As presented in Algorithm 1, SSeeker collects the embedding of one API's SSs to the set X. Initially, it creates the first alive SS group g_1 with the randomly chosen x from X and deletes x from X. Next, it loops every alive group until X is empty. For every alive group g_i, SSeeker averages all the embeddings of g_i as the group embedding c_i and find the most similar embedding x_{sc} and dissimilar embedding x_{sf} between c_i and X by comparing their consine-similarity. Then, SSeeker adopts a greedy strategy to add ungrouped embeddings. If the similarity between the group embedding c_i and its most dissimilar embedding x_{sf} is larger than the preset threshold t, all the embeddings in X are similar with g_i and should be added to it. Then X is empty and the loop would stop. If the similarity between c_i and its most similar embedding x_{sc} is smaller than t, there is no embedding in X similar with g_i so g_i would become dead and not looped anymore. A new group would be created with x_{sc}, which would be deleted from X. Another case is when c_i and x_{sc} are similar, SSeeker would add x_{sc} to g_i and delete it from X. When the looping is finished, all the SSs of this API are divided into several SS groups, which are ready to be reviewed by humans to find the SS issues.

Table 1 Dataset for SS discovery evaluation

Name	API	S	S_{uniq}	S_{sam}	SS	Non-SS
cpp	397	4287	1938	194	30	164
cppref	463	4021	1130	113	21	92
Linux	463	4878	1669	167	26	141
Microsoft	459	4920	2312	200	41	159
Total	463	18,106	7049	674	118	556

Table 2 The effectiveness of SS discovery, compared with keywords-based method

Name	S-HAN				Keyword			
	ACC	F1	FPR	FNR	ACC	F1	FPR	FNR
cpp	0.9	0.69	0.06	0.3	0.85	0.18	0.01	0.9
cppref	0.91	0.77	0.07	0.19	0.84	0.25	0	0.86
Linux	0.92	0.8	0.09	0.04	0.86	0.3	0.02	0.81
Microsoft	0.92	0.8	0.05	0.2	0.82	0.26	0.01	0.85
Average	0.91	0.77	0.07	0.18	0.84	0.25	0.01	0.86

Algorithm 1: Group SSs of one API

```
Function Main(X, t):
    Input: X = {x_1, x_2, ..., x_n}: the set of one API's SS embeddings; t: threshold for
           similarity
    Output: G = {g_1, g_2, ..., g_m}: the set of SS groups
    x ← randomly_choose(X) // Randomly choose one embedding x;
    X.delete(x);
    g_1 = {x} // Create the first group g_1 with x;
    g_1.alive ← True // The status of g_1 is alive;
    G.add(g_1);
    while X is not NULL do
        foreach g_i ∈ G do
            if g_i.alive then
                c_i ← average_embeddings(g_i) // Generate the group embedding by
                  averaging all the embeddings in g_i;
                sc, sf ← closest_farthest(c_i, X) // Locate the indexes of closest and
                  farthest embeddings in X compared with g_i;
                if cos(c_i, x_sf) > t then
                    g_i.add(X);
                    X ← NULL;
                else if cos(c_i, x_sc) < t then
                    g_i.alive ← False;
                    new g = {x_sc};
                    X.delete(x_sc);
                else if cos(c_i, x_sc) > t then
                    g_i.add(x_sc);
                    X.delete(x_sc);
            end
        end
    end
return G
Function closest_farthest(c, X):
    Input: c: one embedding; X: the set of embeddings
    Output: sc, sf: the indexes of closest and farthest embeddings in X compared with c
    score_l, score_s, sc, sf ← 0, 0, 0, 0;
    foreach x_j ∈ X do
        score ← cos(c, x_j);
        if score > score_l then
            score_l ← score;
            sc ← j;
        else if score < score_e then
            score_e ← score;
            sf ← j;
    end
return xc, xf
```

Evaluation

Implementation

Below are the implementation details of the four components of SSeeker.

Sentence completion. We utilized StanfordCoreNLP (Lynten 2018) to perform dependency parsing and POS tagging for consituent completion and resolution of specific coreference, while using AllenNLP (Gardner et al. 2017) to perform traditional coreference resolution of pronouns.

SS discovery. We employed the pre-trained S-HAN from the previous work (The pre-trained S-HAN 2021). The hyperparameters of this model are 300 dimension for word embedding, 50 layers for Bi-GRU with L2 regularization of 1e−8 factor, 100 layers for dense layer with an ReLU activation and L2 regularization of 1e−8 factor, 1 attention layer with normal distribution initialization, Adam optimizer of 0.001 learning rate, and categorical cross-entropy loss strategy. The word embeddings were trained based on 40,000 sentences of Linux manual

pages (man3 2021) using gensim Řehůřek (2021) for 100 iterations with 16 batch size, 2 epoch.

Sentence embedding generation. We chose pre-trained "all-MiniLM-L6-v2" model (SBERT 2021) which was trained on over 1 billion pairs. Its hyperparameters are 384 dimension for embeddings, 256 max sequence length and mean pooling strategy.

SS grouping. We utilized `util.pytorch_cos_sim()` API of SBERT to quickly compute the cosine scores of one SS (query) with other SSs of the same API (corpus).

Experiment setting

Dataset

We chose API reference documents from 4 popular C Standard library online websites (cpp, cppref, Linux, and Microsoft). As mentioned earlier ("C standard library" section), C standard library is not the same under different C language standards, so in order to conduct further experiments, we investigated APIs under the C99 since it is widely used. Note that our approach can be applied to other C standards and other documents easily. For cpp and cppref, we crawled all the webpages and parsed documents to extract API-related information using bs4 (Beautiful Soup Documentation 2021) and lxml (lxml 2021). For Linux, we downloaded the document archive from the official site (Linux man page 2021) and parsed the files under "*man2*" and "*man3*". For Microsoft, we cloned their MicrosoftDocs Github repository (Microsoft 2021b) and parsed Markdown files. We stored API-related information (including API prototypes and documents) if this API belongs to C99 APIs. Note that there are 463 APIs in total after summing all the APIs from different websites, which is also the number of all C99 APIs. Table 1 shows the number of APIs (Column "*API*") and the number of these APIs' sentences (Column "*S*" before removing duplicate data and column "S_{uniq}" after deduplication).

Platform

All the experiments are conducted on an Ubuntu 16.08 with 8 cores CPU (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz), 128G memory, and 3TB hard drivers.

Effectiveness

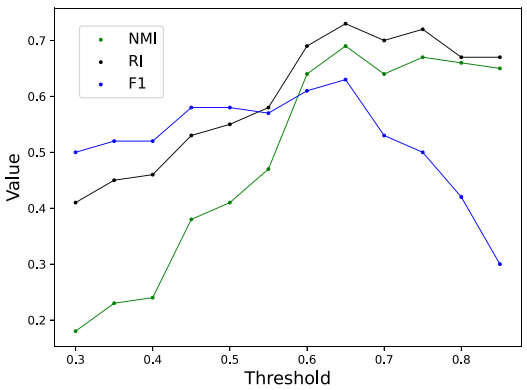
In this part, we would evaluate the effectiveness and performance of SSeeker to answer two research questions.

- **RQ1**: Can SSeeker discover SSs from API documents effectively even if SSs have various syntax structures?
- **RQ2**: Would our sentence embeddings method perform well in the semantic similarity comparison?

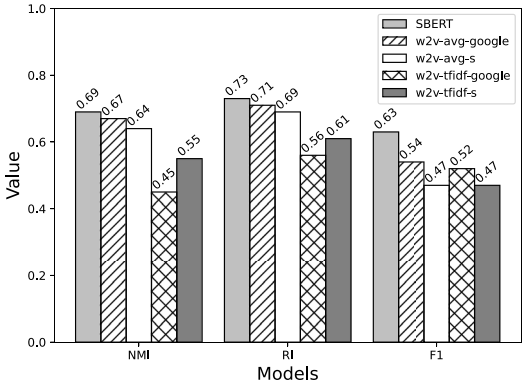
Experiments for answering RQ1

In order to evaluate the effectiveness of discovering SSs, we applied the S-HAN model to sampled sentences of every document source and measure its performance through accuracy, F1, FPR, and FNR metrics. Besides, we compared S-HAN with the previous keyword-based method (Tan et al. 2007) to check if the sentiment analysis-based method outperforms other methods.

Firstly, we applied systematic sampling to the sorted S_{uniq} and sampled around 10% sentences (Column "S_{sam}" in Table 1). If the number of samples is larger than 200, we only choose 200 samples. Then we manually annotated them as SS (Column "*SS*" in Table 1) and non-SS (Column "*non-SS*" in Table 1). Next, we applied S-HAN model to the S_{sam} and achieved an average accuracy of 91.2%, false negative rate of 19.4%, and false positive rate of 6.2%, as shown in Table 2. In addition, the results showed that S-HAN outperforms the keywords-based



(a) Performance of SS grouping when threshold ranging from 0.3 to 0.85



(b) Comparing SBERT with other Word2Vec-based models for SS grouping

Fig. 4 Evaluation of SS grouping

SS discovery method (Tan et al. 2012), especially for the accuracy and F1. Then we ran SS discovery on all the sentences of 4 online document websites and identified 691, 871, 971, and 967 SS for cpp, cppref, Linux, and Microsoft respectively.

Experiments for answering RQ2

To measure the effectiveness of our sentence embedding method, we first evaluated the performance of SS grouping on sampled APIs using metrics of clustering (NMI, RI, F1). Then we compared the SBERT model with other sentence embedding methods based on the Word2Vec model.

Firstly, in order to evaluate the effectiveness of SS grouping, we first applied SS discovery to all the S in D_{doc} and collected the common APIs (168) with SS among 4 documentation websites. Then we applied systematic sampling to the sorted common APIs and sampled 13 APIs. These APIs have 125 SSs and 54 SS groups (D_{ssg}) in total. Next, we chose the optimal similarity threshold t by setting the threshold in the range [0.3, 0.85] and measuring the performance of SS grouping. Since the SS grouping is similar to clustering, we chose the external criterions of clustering as metrics. The set of SSs to be clustered has N SSs; SS group generated by SS grouping is the cluster and its label is assigned to the most common class in the group; the manually labelled D_{ssg} is the benchmark or gold standard. Here are the criteria we used (Wikipedia 2021h; NLP 2021).

- *Normalized Mutual Information (NMI).* It measures the mutual dependence between the two variables. As the normalized MI, NMI ranges between 0 (no mutual information) and 1 (perfect correlation).
- *Rand Index (RI).* RI measures the percentage of correct decisions. Its equation is:

$$RI = \frac{TP + TN}{TP + FP + FN + TN} \tag{2}$$

  where TP, TN, FP, and FN have the same meaning as in the classification task. From this perspective, RI is to clustering as the accuracy is to classification.
- *F-1 measure (F1).* F1 is the weighted average of precision ($TP/(TP + FP)$) and recall ($TP/(TP + FN)$), which also has the same meaning as in classification.

Using these criteria, SS grouping performs best under the threshold 0.65, achieving an average NMI of 69%, RI of 73%, and F1 of 63%, which can be seen in the Fig. 4a.

Furthermore, we compared the SBERT sentence embedding model with another state-of-the-art word embedding model Word2Vec. Since Word2Vec only provides embeddings for words, we adopted two popular methods to compute sentence embeddings respectively: the first method averages all the word embeddings (we call it "w2v-avg" for short) and the second method uses "tf-idf" (Wikipedia 2021i) as the weight of word embeddings (we call it "w2v-tfidf" for short). For the Word2Vec model, we used the pre-trained model (Google 2021) on Google News Corpus or the new model trained on the C library document sentences S of D_{doc} respectively. The hyperparameters for newly trained Word2Vec model are iteration 50, word embedding dimension 128, and the windows size 3. Thus the four models compared with SBERT are "w2v-avg-google", "w2v-avg-s", "w2v-tfidf-google", and "w2v-tfidf-s", where "-google" means using Google pre-trained model and "-s" means using our trained model. Their thresholds are determined separately and optimal for each model. The comparison results are shown in Fig. 4b. Our study shows that SBERT performs better than other models, especially in the F1 score.

Performance

We ran SSeeker on the 4 online C documents (1.91MB files), the first two components took 1,796 seconds. The time cost of SS grouping varies with the number of SSs and the average cost is 60 seconds per API (186 SSs). We can conclude that SSeeker is quite fast and efficient.

Discoveries

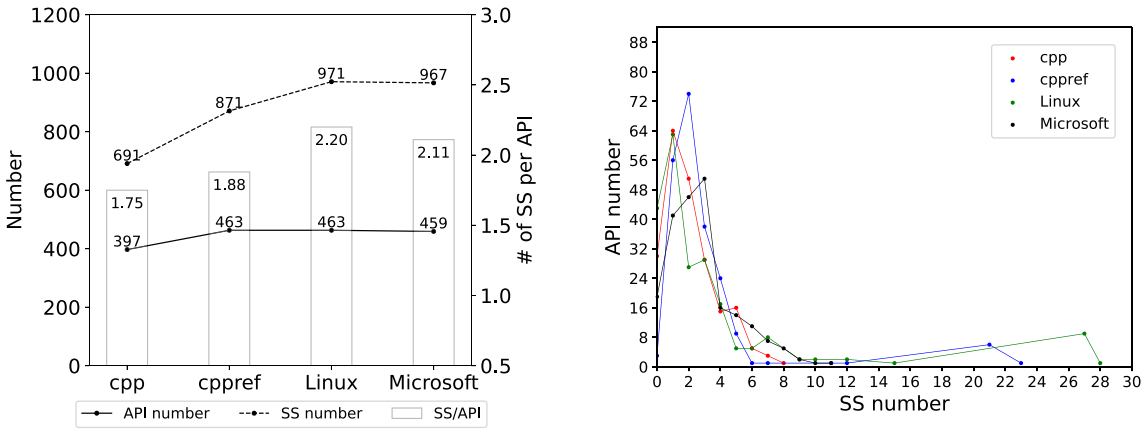
With the help of SSeeker to find the missing and inaccurate SS, we deeply investigated into 4 popular online C Standard libraries API documentation and revealed 3 types of documentation issues. Lastly, we proposed 3 suggestions for documentation maintainers.

Findings of the online C standard library documents

The conformity with the C99 standard

Not all the popular online C standard library documents conform with C99 standard completely and accurately neither as developers thought of or as documentation maintainers declared. According to C99 Standard official document (ISO 2021a), there are 24 header files and 463 functions (not considering macros). However, as shown in Table 1, only cppref and Linux provide all the C99 APIs but they also have API prototype and SS issues as described in the following.

Although cpp says that they support C90 and C99 in the "*Note on versions*" part of the introduction of the C library, their displayed C99 APIs are not complete and accurate. Compared with 24 headers and 463 APIs of C99 standard, cplusplus only provide 23 headers and 397 APIs, while lacking *complex.h* header and 66 APIs, e.g., `cacos`. Cppref is the most comprehensive documentation among 5 online documents. It contains API



(a) API number, SS numbers and the ratio of SS per API

(b) The number of APIs with different number of SSs

Fig. 5 Comparing the APIs, SS and the ratio of SS per API

documents for different C standard libraries (C90, C95, C99, C11, and C17). Linux man page also provides all the C99 APIs, but they don't distinguish between different versions of APIs. For example, according to the "*CONFORMING TO*" part of the webpage(man7 2021b) of `strcat`, this document conforms to C89 and C99. However, the provided prototype only includes the C99 version "`char *strcat(char * restrict dest, const char * restrict src);`" not the C89 version "`char *strcat( char *dest, const char *src );`". Although there are no explicit notes about the supported standards in every API's web page, Microsoft says that they basically support C99, while missing the implementation of some types and APIs. From Table 1, we can see Microsoft provide 459 C99 APIs, while lacking 4 APIs, which are unimplemented (`copysignf` and `copysignl`), obsoleted (`gets`), or just forgotten to display the prototype (`truncf`) on the webpage.

APIs and SSs of the documentation

In this part, we further inspected the basic information about APIs and their SSs every documentation website provides.

From Table 1, we could find that cppref, Linux, and Microsoft provide most of the C99 APIs (more than 450 APIs). Besides, the common intersection of 4 documents are 397 APIs, mostly limited by cpp documentation.

Next, we studied the number of APIs, SSs and the ratio of SS per API, shown in Fig. 5a. We found that every API has around 2 SSs on average, which is the same in 4 documentation websites. Then we concentrated on the common 397 APIs and studied the distribution of SSs per API, shown in Fig. 5b. We discovered that nearly all the APIs in cppref have their SSs, and only 3 APIs has no SS. Most APIs have 0, 1, 2 or 3 SSs, which is consistent with the previous observation that averages SSs per API is about 2. There is an anomaly that cppref and Linux have several APIs possessing more than 20 SSs. We looked into that and found these APIs are `printf`, `vfprintf`, and other APIs related to formatted data. Their documents include not only the basic information of these APIs but also the syntax of conversion specifications, which is very long and contains lots of SSs.

Issues of API prototype

After the inspection of these documents' conformance with the C99 standard, we continued to look into the API name, parameter types, and return type to check if prototypes of these APIs are the same as proclaimed in the C99 standard. We found 92 prototype issues through comparing online documents with C99 standard documents. Here are the main types and examples.

- *Mishandled type qualifier.* Type qualifier is a keyword that is applied to a type, which will turn into a *qualified type*. We found that type qualifiers *const* and *restict* are often mishandled. Cpp lacks *const* in 9 APIs prototype. For example, the prototype of `strrchr` in cpp is "`char * strrchr ( char * str, int character );`", while the correct one should be "`char * strrchr (const char * str, int character );`". Even though the webpage of `strrchr` in cpp also provides another prototype "`const char * strrchr (const char * str, int character );`", it's still not right. On the opposite, Microsoft puts an extra *const*

in the prototype of `_Exit` and `exit`. For keyword *restict*, cpp misses it in 74 APIs (e.g., `fopen`), cppref only misses it in `wcsftime` and Linux only misses it in `strcpy`. Microsoft doesn't provide *restict* type qualifier but implements its specific *__restict* keyword. There are 87 prototype issues of this type: 74 issues of *restict* for cpp, 1 issues of *restict* for cppref, 1 issue of *restict* for Linux, 9 issues of *const* for cpp, and 2 issues of *const* for Microsoft.

- *Wrong type or API name*. Except for the missing or extra type qualifier, parameter types and return types can also be wrong. The return type of `nanl` in cpp should be `long double` not `float`. The return type of `towupper` and `towlower` in Microsoft should be `wint_t` not `int`. Besides, the API name of `wcstoll` prototype in cpp is mistakenly written as `strtoll`. There are 4 prototype issues of this type as shown above.
- *Missing prototype*. It can happen that one API has its document but not its prototype on its web page. For example, the web page of `truncf` in Microsoft has a "*Syntax*" part, which describes the prototypes of other APIs on this page but misses the one of `truncf`.

Issues of API SS

After manually analyzing the discovered missing and inaccurate SSs, we found 96 SS issues and summarized 7 common types of SS issues. 96 SS issues include 1 inaccuracy issue and 95 absence issues. These absence issues cannot be found by previous work because they are not designed to detect missing SSs.

- *Array*. Important SSs for array include "*should have enough size*", "*should not overlap*", and "*should not be NULL pointer*". However, these SSs are often missing. For example, `memmove` in Linux lacks "*should have enough size*"; `strncat` in cpp lacks "*should not overlap*"; `memcpy` in cpp lacks "*should not be NULL pointer*'. During the analysis, we found an inaccurate SS of `memcpy` in Microsoft Docs. The SS is "*Make sure that the destination buffer is the same size or larger than the source buffer.*", but it doesn't matter even if the destination buffer is smaller than the source buffer when the destination buffer is larger than the count (number of characters to copy). This issue has been confirmed (Zero0one1 2021) by the documentation maintainers. Lacking this type of SSs may cause programmers to write code with bug, e.g., buffer overflow. There are 39 SS issues of this type: 11 for cpp, 2 for cppref, 15 for Linux, and 11 for Microsoft. 1 SS issue of Microsoft is inaccuracy and the others are missing issues.
- *Pointer*. In special cases, pointers shall not be dereferenced. For example, if the parameter of `malloc` is zero, the return value may be a non-NULL pointer but shall not be dereferenced, as described in cpp and cppref. Lacking this type of SSs may cause NULL pointer dereference vulnerability. There are 2 SS missing issues of this type: 1 for Linux and Microsoft respectively.
- *String*. It is important to make sure the string is valid, i.e., C-string ending with a terminating "null character", during the reading or writing operations. The documents of `atoi` in cpp and cppref stress this SS, while Linux and Microsoft miss it. If the programmer doesn't follow this SS, overrun or NULL pointer dereference can occur, e.g., CVE-2018-14884 in PHP. There are 30 SS missing issues of this type: 12 for cpp, 2 for cppref, 14 for Linux, and 2 for Microsoft.
- *API sequence*. This type includes the specification of parameters or the return value and the specification of the API call sequence. The former type of SS looks like "*The parameters or return value should (not) be called with another API before or after you call this API.*". One example of the latter type is the SS of `quick_exit` in cpp, "*If a program calls both exit and quick_exit, or quick_exit more than once, it causes undefined behavior.*". Lacking this type of SSs may cause memory leaks or system crashes. There are 14 SS missing issues of this type: 4 for cpp, 1 for cppref, 4 for Linux, and 5 for Microsoft.
- *Data type range*. The range of basic data type is limited and the behavior would be undefined if the converted data is out of the range of representable values by basic data type, e.g., `atoi` and `abs`. There are 6 SS missing issues: 4 for Linux and 2 for Microsoft.
- *Direct modification*. The return value of some APIs should not be modified, otherwise cause undefined behavior, e.g., `getenv`. Lacking this type of SSs may cause buffer overflow vulnerability. Cpp, Linux, and Microsoft lack 1 SS of this type respectively.
- *Suggestion*. The SS suggestions are not compulsory but can make the application more secure. For example, `rand` is "*not recommended for serious random-number generation needs, like cryptography*" as noted in cppref, while the other three document websites lack it.

Issues of web page display

The web pages of API documentation have the following display issues, which can make readers confused and should be improved.

- *Unclear support for different standards*. As mentioned in "*The Conformity with C99 Standard*" ("The conformity with the C99 standard" section), most of the online documentation websites don't have a clear, correct, and integrated indication web page about their supported C standards. They might provide an outdated blog or scattered blogs explaining their support for different features. It brings quite hard work for programmers, who want to use APIs correctly, to search standard related information in their official websites or search engines, which may turn out to be no search result or inaccurate result.
- *Unobvious distinction of different standards*. When online C standard library documentation supports more than one standard, it may ignore the difference between different versions of the API. For one thing, the documentation may only give one prototype of the API and does not mark its standard; for another thing, the SSs might be mixed together with no matter which standard its API belongs to. This unobvious distinction gives the programmer inaccurate information and probably wrong guidance. Among these 4 websites, the cppreference website provides the clearest and most obvious distinction in the prototypes and SSs.
- *Incomplete and inconsistent displayed information*. The information displayed on the web pages can be incomplete and inconsistent. Online C standard library documentation (e.g., cppref, Linux, and Microsoft) often offers a web page showing the alphabetic list of APIs it provides. But the content may be incomplete and inconsistent with all the APIs it actually provides. Microsoft lacks `expl` and other 9 APIs in the API list. Other display incompleteness includes missing API name in the web page header (e.g., `fdimf` in cpp), the incomplete web page display (e.g., *setjmp.h* web page does not have the reference section on its bottom right corner).

Suggestions for documentation maintainers

According to the above findings, we propose several suggestions for documentation maintainers. Among these, the second and the third suggestions are not limited to the C standard library but also apply to other libraries.

- *Clearly depicting the supported standards*. First, documentation maintainers should depict its supported standards clearly and accurately on the home web page of the document introduction. If only supporting certain features or having its specific implementation, they also should describe it at the same time. Secondly, API documents should distinguish the API prototypes and SSs if they differ among standards as cppref documentation does.
- *Providing comprehensive SSs*. The documentation maintainers should provide comprehensive SSs to guide developers securely using APIs. Based on our summary of frequently missing SSs, documentation maintainers of C language software should pay more attention to the SSs related to the array, pointer, string, API call sequence, direct modification, and secure suggestions.
- *Maintaining the consistency between code and documentation*. The consistency between code and documentation is significant during the development and maintenance of software, which can be seen from our research. Microsoft implements its C runtime library, which includes most of C standard library APIs and Microsoft-specific APIs. However, there are inconsistencies between the code and the online documentation, e.g., the return type of `towupper` is `win_t` in the code implementation but the online prototype says it's `int_t`, as mentioned in "Issues of API prototype" section. Keeping the consistency between code and documentation can reduce the documentation issues and lower the cost of development and maintenance.

Discussion

Limitations. With the high accuracy of 91.2%, SS classification still introduces false data affecting the result and the analysis in "APIs and SSs of the documentation" section. Besides, SSeeker cannot effectively group complex SSs, which contain more than one constraint meaning or that contain clauses that are not related to constraints. For example, an SS containing two constraint meanings is semantically similar to two SS groups but it can only be divided into one group according to our algorithm. In addition, SSeeker considers all words of the sentence to generate its embedding, but there may be phrases that are irrelevant to the content of the constraint thus introducing noise into the embedding.

Specific implementations of C standard library. Linux and Microsoft don't simply provide documentation for C standard library APIs like cpp and cppref. They also implement their own C runtime libraries ("glibc" for Linux, "MSVCRT" and "UCRT" for Microsoft) and may bring in a few changes which make their API documents different from the original C standard library API documents. For example, Linux brings in a new type `sighandler_t` for `signal` API and Microsoft does not support C99 *restrict*. Despite these differences, it is still meaningful to study their conformity with the C

standard and useful to discover missing SSs through the help of existing ones.

The reason for not choosing official documentation. We didn't compare online documentation with the official C standard files to find the missing SSs because the official files only supply a part of SSs. Specified by a committee after years of discussion, official C standard documentation is tidy, precious, concise, and time-consuming. Unofficial online documentation is based on official documentation and is open source so it can be updated in time and take into consideration SSs which programmers often violate more freely. For example, the `memcpy` in cppref has an SS "*The behavior is undefined if either dest or src is a null pointer.*", while official documentation does not. Based on this observation, we chose to compare unofficial online documentation to discover more SSs.

Future work.

We will first address the above limitations by breaking sentences into finer granularity to improve the performance of SSeeker. Furthermore, we will study the inconsistency between API documentation and a larger range of text sources (e.g., questions and answers in the StackOverflow and bug reports). These text are numerous, easily accessible for most popular libraries and contain SSs that developers may overlook and thus probably are missing or inaccurate in the API documentation. In addition, these newly found SSs can be used to detect API misuse vulnerability, which could find more bugs than other detection methods based solely on API documentation.

Related work

In this part, we first discussed the related work of studying documentation issues and then talked about previous research related to the two components of SSeeker.

Study of documentation issues.

Previous work discover the documentation issues mainly through detecting the inconsistency between code and descriptive text (API documentation or code comment). Tan et al. (2007) chose comments related to topic "keywords" (e.g., "lock") of C language software when the sentences have imperative words. Their solution iComment then utilized a decision tree to map the comments to rule templates, which were used to generate a state machine to detect rule violation of code. iComment detected 27 bad comments related with "lock" and "call" topic. Another work of detecting comment-code inconsistencies, tComment (Tan et al. 2012) targeted the NULL-related API misuse or wrong comment. It used simple pattern matching to extract properties from Javadoc comments for a method and then generated random tests to check properties and report bugs or wrong comments. Blasi et al. (2018) proposed Jdoctor, which translated Javadoc comments to procedure specifications using pattern matching, lexical matching, and semantic matching. It chose the first successful result of the three approaches. For the semantic matching, Jdoctor used Glove to generate word embeddings and Word Mover's Distance (WMD) algorithm to compare multiple words together. Although it does not aim to find comment issues at first, the authors reported six types of inconsistencies after analyzing Jdoctor's output. Zhong and Su (2013) combined natural language and code analysis technology to detect Java API documentation errors, but their detection range is limited to syntax errors and broken code names. These work listed above can not detect missing SSs but only wrong SSs, whose types are also limited by their text-parsing approaches. DRONE, proposed by Zhou et al. (2020), is an extended version of previous work (Zhou et al. 2017). DRONE parsed the abstract syntax tree of Java code and generated code constraint first-order logic (FOL) with the help of control-flow analysis while generating document constraint FOL using dependency parsing and pattern analysis. Then an SMT solver was applied to report four types of document defects. As far as we know, there is no work studying the documentation issues (including API prototype issues and SS issues) of online C standard library documents.

Classification of security specification.

As for the problem of classifying the security specification sentences in documents, most of the previous work used fixed keywords (Tan et al. 2007) or template matching (Pandita et al. 2012; Zhong et al. 2009; Chen et al. 2019). Tan et al. (2007) summarized frequently appearing keywords in SS, such as "*must*" and "*need to*", to determine whether a sentence is an security specification. Template matching uses part of speech analysis technology in the NLP field to determine whether a sentence meets certain sentence requirements based on a preset shallow parsing (Sha and Pereira 2003) template. Tan et al. (2012) used simple pattern matching to decide three types of NULL-related comments. Toradocu (Goffi et al. 2016) did not classify SS but tried to directly translate the sentence using pattern matching and lexical matching. Similarly, Jdoctor (Blasi et al. 2018) directly translated comments but added semantic similarity analysis to this step. Lv et al. (2020) trained the bidirectional GRU model with attention capturing the emotional tone of SS to complete the judgment, which outperformed keywords-based and template-based methods and could discover more types of SSs.

Semantic similarity of text.

Previous studies on semantic similarity of text can be divided into the corpus-based approach and knowledge-based approach (Chandrasekaran and Mago 2021). Knowledge-based similarity methods calculate the

similarity based on the information derived from knowledge sources (Mikolov et al. 2013), e.g., WordNet (University 2021) which is a popular lexical database of English synonyms. Corpus-based approaches measure the similarity using the information from large corpora with the distributional hypothesis that similar words frequently appear together. Over these years, approaches based on word embedding have been promoted, including Word2Vec (Mikolov et al. 2013), Glove (Pennington et al. 2014), fastText (Bojanowski et al. 2017). With the Transformer structure, Bidirectional Encoder Representations from Transformers(BERT) (Devlin et al. 2019) performed well in multiple NLP tasks, including question answering and classification. Based on BERT, Sentence-BERT (SBERT) (Reimers and Gurevych 2019) was proposed especially for sentence similarity comparison and outperformed other sentence embedding methods, including tf-idf, averaging Glove word vectors, and averaging BERT word vectors and etc.

Conclusion

In this paper, we investigate the popular online websites for C standard library documentation, study their conformity with the C99 standard, and discover their documentation issues. More specifically, we propose an approach SSeeker to help quickly find missing or inaccurate SSs through classifying SSs using sentiment analysis and grouping semantically similar SSs. SSeeker can study more types of SSs and find missing SSs compared with previous work. We analyzed 4 popular online websites and found 92 prototype issues, 15 web page issues, and 96 SS issues. We provide several suggestions for documentation maintainers correspondingly. This study reveals the status quo of C standard library documentation maintained by third-parties and enhances their documents.

Acknowledgements
We would like to thank the anonymous reviewers for detailed comments and useful feedback.

Authors' contributions
Ruishi Li: investigation, conceptualization, methodology, materials, writing, editing, experiment, validation, review, resources. Yunfei Yang: resources, discussion, experiment, review. Jinghua Liu: discussion, experiment, review. Peiwei Hu: discussion, experiment, review. Guozhu Meng: discussion, review, supervision. All authors read and approved the final manuscript.

Funding
Our research was supported in part of the National Key Research and Development Program (No. 2020AAA0104301), National Natural Science Foundation of China (No. U1836211, 61902395), the Anhui Department of Science and Technology (No. 202103a05020009) and Beijing Academy of Artificial Intelligence (BAAI).

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.

Received: 4 January 2022 Accepted: 21 February 2022
Published online: 02 July 2022

References

Beautiful Soup Documentation (2021) https://www.crummy.com/software/BeautifulSoup/bs4/doc/
Blasi A, Goffi A, Kuznetsov K, Gorla A, Ernst MD, Pezzè M, Castellanos SD (2018) Translating code comments to procedure specifications. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis
Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. Trans Assoc Comput Linguist 5:135–146
Chandrasekaran D, Mago V (2021) Evolution of semantic similarity-a survey. ACM Comput Surv (CSUR) 54:1–37
Chen Y, Xing L, Qin Y, Liao X, Wang X, Chen K, Zou W (2019) Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In: USENIX security symposium
Chung J, Gulcehre C, Cho K, Bengio Y (2014) Empirical evaluation of gated recurrent neural networks on sequence modeling
cplusplus: cplusplus (2021) http://www.cplusplus.com/
cplusplus: nanl (2021). https://www.cplusplus.com/reference/cmath/nanl/?kw=nanl
cppreference: cppreference (2021) https://en.cppreference.com/
Devlin J, Chang M-W, Lee K, Toutanova K (2019) Bert: pre-training of deep bidirectional transformers for language understanding. In: NAACL
Gardner M, Grus J, Neumann M, Tafjord O, Dasigi P, Liu NF, Peters M, Schmitz M, Zettlemoyer LS (2017) Allennlp: a deep semantic natural language processing platform. arXiv:1803.07640
Goffi A, Gorla A, Ernst MD, Pezzè M (2016) Automatic generation of oracles for exceptional behaviors. In: Proceedings of the 25th international symposium on software testing and analysis
Google: GoogleNews-vectors-negative300.bin.gz (2021) https://drive.google.com/file/d/0B7XkCwpI5KDYNINUTTlSS21pQmM/edit?usp=sharing
Group, T.S.N.L.P. (2021) Stanford Log-linear Part-Of-Speech Tagger. https://nlp.stanford.edu/software/tagger.shtml
Group, T.S.N.L.P. (2021) Stanford deterministic coreference resolution system. https://nlp.stanford.edu/software/dcoref.shtml
ISO: ISO/IEC 9899:2018 (C17 and C18) (2021) https://www.iso.org/standard/74528.html
ISO: N1256 (C99) (2021) http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
Kim Y (2014) Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882
Lai S, Xu L, Liu K, Zhao J (2015) Recurrent convolutional neural networks for text classification. In: 29th AAAI conference on artificial intelligence
Linux man page (2021) https://man7.org/linux/posix-man-pages/
Liu B, Meng G, Zou W, Li F, Gong Q, Lin M, Sun D, Huo D, Zhang C (2020) A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In: 2020 IEEE/ACM 42th international conference on software engineering (ICSE), pp 1547–1559
Lv T, Li R, Yang Y, Chen K, Liao X, Wang X, Hu P, Xing L (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
Lynten: stanfordcorenlp (2018) https://github.com/Lynten/stanford-corenlp
lxml (2021) https://lxml.de/
man3 (2021) https://linux.die.net/man/3/
man7: man page (2021) https://man7.org/linux/man-pages
man7: strcat (2021) https://man7.org/linux/man-pages/man3/strcat.3.html
Microsoft: Microsoft documentation Github repository (2021) https://github.com/MicrosoftDocs/cpp-docs/blob/master/docs/c-runtime-library/reference
Microsoft: Microsoft documentation (2021) https://docs.microsoft.com/en-us

Mikolov T, Chen K, Corrado GS, Dean J (2013) Efficient estimation of word representations in vector space. In: ICLR
NLP S (2021) Evaluation of clustering. https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html
(NVD), N.V.D.: CVE-2005-3346 (2021) https://nvd.nist.gov/vuln/detail/CVE-2005-3346
Pandita R, Xiao X, Zhong H, Xie T, Oney S, Paradkar A (2012) Inferring method specifications from natural language api descriptions. In: 2012 34th international conference on software engineering (ICSE), pp 815–825
Pennington J, Socher R, Manning CD (2014) Glove: global vectors for word representation. In: EMNLP
Řehůřek R (2021) gensim . https://radimrehurek.com/gensim/
Reimers N, Gurevych I (2019) Sentence-bert: sentence embeddings using siamese bert-networks. ArXiv:abs/1908.10084
SBERT: all-MiniLM-L6-v2 model (2021) https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
Sha F, Pereira F (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, pp 213–220. https://www.aclweb.org/anthology/N03-1028
StanfordParser (2016) https://nlp.stanford.edu/software/dependencies_manual.pdf
Tan L, Yuan D, Krishna G, Zhou Y (2007) /*icomment: bugs or bad comments?*/. In: SOSP
Tan SH, Marinov D, Tan L, Leavens G (2012) @tcomment: testing Javadoc comments to detect comment-code inconsistencies. In: 2012 IEEE 5th international conference on software testing, verification and validation, pp 260–269
The pre-trained S-HAN (2021) https://github.com/lvtao-sec/Advance/tree/master/S-HAN/saved-models
University P (2021) WordNet. https://wordnet.princeton.edu/
Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, pp 5998–6008
Wikipedia: C standard library (2021) https://en.wikipedia.org/wiki/C_standard_library
Wikipedia: The ISO/IEC 9899 Standard (2021) https://www.iso-9899.info/wiki/The_Standard
Wikipedia: ANSI X3.159-1989 (C89) (2021) https://en.wikipedia.org/wiki/ANSI_C#C89
Wikipedia: ISO/IEC 9899:1990 (C90) (2021) https://en.wikipedia.org/wiki/ANSI_C#C90
Wikipedia: ISO/IEC 9899:1990/AMD1:1995 (C95) (2021) https://en.wikipedia.org/wiki/ANSI_C#C95
Wikipedia: ISO/IEC 9899:1999 (C99) (2021) https://en.wikipedia.org/wiki/ANSI_C#C99
Wikipedia: ISO/IEC 9899:2011 (C11) (2021) https://en.wikipedia.org/wiki/ANSI_C#C11
Wikipedia: Cluster analysis (2021) https://en.wikipedia.org/wiki/Cluster_analysis#External_evaluation
Wikipedia: tf-idf (2021) https://en.wikipedia.org/wiki/Tf-idf
Yang Z, Yang D, Dyer C, He X, Smola A, Hovy E (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, pp 1480–1489
Yu D, Yang G, Meng G, Gong X, Zhang X, Xiang X, Wang X, Jiang Y, Chen K, Zou W, Lee W, Shi W (2021) SEPAL: Towards a large-scale analysis of SEAndroid policy customization. In: Proceedings of the 30th The Web Conference (WWW)
Zero0one1: Microsoft SS Issue (2021) https://github.com/MicrosoftDocs/cpp-docs/issues/3366
Zhong H, Zhang L, Xie T, Mei H (2009) Inferring resource specifications from natural language API documentation. IEEE/ACM Int Conf Autom Softw Eng 2009:307–318
Zhong H, Su Z (2013) Detecting api documentation errors. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications
Zhou Y, Wang C, Yan X, Chen T, Panichella S, Gall HC (2020) Automatic detection and repair recommendation of directive defects in java API documentation. IEEE Trans Softw Eng 46:1004–1023
Zhou Y, Gu R, Chen T, Huang Z, Panichella S, Gall HC (2017) Analyzing apis documentation and code to detect directive defects. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 27–37

Publisher's Note