Yin *et al. Cybersecurity* (2024) 7:30
https://doi.org/10.1186/s42400-024-00224-w

Cybersecurity

RESEARCH

Open Access

Discovering API usage specifications for security detection using two-stage code mining

Zhongxu Yin[1*], Yiran Song[2†] and Guoxiao Zong[1*†]

Abstract

An application programming interface (API) usage specification, which includes the conditions, calling sequences, and semantic relationships of the API, is important for verifying its correct usage, which is in turn critical for ensuring the security and availability of the target program. However, existing techniques either mine the co-occurring relationships of multiple APIs without considering their semantic relationships, or they use data flow and control flow information to extract semantic beliefs on API pairs but difficult to incorporate when mining specifications for multiple APIs. Hence, we propose an API specification mining approach that efficiently extracts a relatively complete list of the API combinations and semantic relationships between APIs. This approach analyzes a target program in two stages. The first stage uses frequent API set mining based on frequent common API identification and filtration to extract the maximal set of frequent context-sensitive API sequences. In the second stage, the API relationship graph is constructed using three semantic relationships extracted from the symbolic path information, and the specifications containing semantic relationships for multiple APIs are mined. The experimental results on six popular open-source code bases of different scales show that the proposed two-stage approach not only yields better results than existing typical approaches, but also can effectively discover the specifications along with the semantic relationships for multiple APIs. Instance analysis shows that the analysis of security-related API call violations can assist in the cause analysis and patch of software vulnerabilities.

Keywords Specification mining, Frequent API sequence, Semantic relationship, Under-constrained symbolic execution, Vulnerability mining

Introduction

In programming, calls to application programming interface (API) functions usually need to follow a particular specification. These specifications generally express the intrinsic characteristics and security requirements of the program. Program developers who fail to adhere to these specifications can easily introduce defects into the program, which may even cause serious security problems. In SSLINT (He et al. 2015), the author manually built the API security usage specifications of the certificate validation process in the open-source projects OpenSSL and GnuTLS, and found a dozen or so lack of verification errors in the source code that could lead to man-in-the-middle attacks. Unfortunately, although some APIs give formal usage specifications in the official documentation, many APIs do not have public usage specifications (Jana et al. 2016). Because the same set of APIs may be widely used, the vulnerability caused by violations of

†Yiran Song and Guoxiao Zong have contributed equally to this work.

*Correspondence:
Zhongxu Yin
yinzhxu@163.com
Guoxiao Zong
zonegoalshall@foxmail.com
[1] Information Engineering University, Zhengzhou 450001, China
[2] Henan University of Animal Husbandry Economy, Zhengzhou 450046, China

the specification is highly likely to be reproduced in different applications. Therefore, the mining of API usage specifications in a program has become an important aspect of program security analysis.

Some methods automatically extract API usage specification and use the specifications for security analysis. To mine the association relationships of APIs, PR-Miner (Li and Zhou 2005) uses an effective frequent itemset mining technique. Frequent itemset mining is a branch of data mining that focuses on looking at sequences of actions or events, each of which have a number of features. The aim of a frequent itemset mining algorithm is to find all common sets of items, defined as those itemsets that have at least a minimum support (Tamaskar and Raut 2016). However, the rules extracted by PR-Miner are redundant and lack of parameters of and conditional dependencies between APIs.

For other mining methods such as frequent subgraph mining (Huan et al. 2004) and under-constrained symbolic execution (Ramos and Engler 2015), the semantic beliefs are included. Instead of performing symbolic execution from the start position of the program, under-constrained symbolic execution assumes a certain precondition and starts analysis directly from the entry of the target function of interest. Even so, the computing cost is relatively large and cannot be scaled to large programs. There are still plenty of complex functions in large and complex programs, the overhead is still unaffordable.

In this paper, our aim is to use an API specification mining approach that efficiently extracts a relatively complete list of the API combinations and semantic relationships between APIs. The extracted API specifications and security-sensitive function model (Yin et al. 2020) constitute the static method to detect the call sequences that violates the API specification and for bug discovery and repairment.

The existing methods of extracting the relationship between APIs are usually limited to the relationship between pairwise APIs. Our method extracts the relationship in all related API sequences, and then recover the API call specifications without documents.

To achieve the goal above, we proposed an approach to analyze the target program in two phases. The first phase extracts context-sensitive frequent API sequences using frequent itemset mining. The second phase takes the sequences mined in the first phase and mines the specifications that contain the semantic relationships of multiple APIs using under-constrained symbolic execution.

The main contributions of this work are:

1. A new API specification mining approach that can automatically and efficiently extract API specifications containing semantic relationships for multiple APIs. The approach has two features.
   (a) Mainly from source code. The approach can mine specifications mainly from software code without any prior knowledge about the software or requiring any rule templates, annotation, or feedback from programmers.
   (b) Domain adapted. Our method is able to deal with longer and relatively complete API sequences. Meanwhile, path pruning based on domain adapting can obtain better extraction efficiency and scalability.

   The experimental results show that the approach can mine usage specifications that contain the semantic relationships among multiple APIs. Moreover, the accuracy and efficiency are clearly better than those of the existing typical approaches.
2. We proposed the concept of frequent common API in the problem domain of frequent API sequence mining. The structural characteristics of frequent common APIs in the FP tree are proposed and engaged with an accurate identification and filtration method of frequent common API, which greatly improved the effectiveness of our proposed frequent API maximum sequence mining algorithm**.** An FP tree is the main structure for representing itemsets in the frequent closed-itemset mining algorithm FPclose(Grahne and Zhu 2003a). Some APIs are called frequently, but they are mainly common functions that implement error message prompts after an error. Such APIs are referred as frequent common APIs in this paper. The algorithm identifies and filters out the frequent common API nodes in the FP tree that are close to the root node and have a large overall frequency and out degree, which reduces their interference on the mining results.
3. We implemented the proposed approach and made evaluations. The results show that the proposed semantic relationship specification mining approach is effective and superior to relevant approaches. Further analysis on violations of the specifications reveals various security problems.

The remainder of this paper is organized as follows: Sect. "Related work" presents the related work. Sect. "Problem definition" analyzes the key issues of this paper for specification mining through a motivating example. sect. "Semantic relationship sensitive API specification mining approach" describes the overall approach

of our method, the improved maximal frequent itemset mining algorithm, and the semantic relationship extraction and call specification mining of APIs. Sect. "Implementation and evaluation" describes the implementation and presents the evaluation results. Sect. "Analysis of API call specifications violations" states the typical case. Sect. "Conclusion" concludes our work and discusses directions for future work.

Related work

API specification mining uses source code mining approaches to discover the conditions, calling sequences, and semantic relationships of API calls in a program (Dyer et al. 2013). After more than ten years of efforts by researchers and the development of program analysis technology, research in this area has gained some achievements. Typical examples include frequent itemset mining-based approaches, security-sensitive function-based mining approaches, template-based mining approaches and document-based method.

Security-sensitive function-based mining approach

In the security-sensitive function-based mining approach, security-sensitive functions (Chen et al. 2018) are discovered. The specifications are then revealed by mining the pre-and post-conditions of these functions (Nguyen et al 2015, 2014; Ramanathan et al. 2007)) Liang et al. proposed AntMiner ((Bian et al. 2018a; Liang et al. 2016), which uses the idea of program slicing to preprocess the source code and reduce noise interference. It then finds security-sensitive functions through a heuristic approach and computes their preconditions to yield the specification. The Chucky approach (Yamaguchi et al. 2013) proposed by Yamaguchi et al. uses a manually specified sensitive function or variable as a starting point. It then slices statements in the calling functions of a sensitive function and clusters the conditional statements in the slice, outputting the analysis result as a usage specification for the sensitive function. The APEX approach (Kang et al. 2016) proposed by Yuan et al. analyzes the post-conditions of each API function called by the program, finds the fallible APIs that are sensitive to error handling, and identifies error paths and non-error paths according to the number of branching points of the path to find the error return values that handle specification. The approach proposed by Chang (Chang et al. 2008, 2012) first chooses the set of APIs of interest, then for every API, uses each of its call site instances to construct dependence spheres from a system dependency graph of the target program. It then performs frequent isomorphic graph minor mining from the dependence spheres. The frequent isomorphic graph minors are selected as the usage specification. This approach can discover the calling order and semantic relationships of multiple APIs in a program.

The security-sensitive function-based mining approaches implement path-sensitive and flow-sensitive specification mining for specific functions but must first locate sensitive functions. Their effectiveness depends on the accuracy of security-sensitive function identification, and they cannot discover specifications in APIs that are not recognized as sensitive functions. Moreover, the relationships for more than two functions cannot be obtained. In our approach, we filter unrelated APIs and retain the other APIs instead of just choosing candidate APIs for specification mining. This enables us to have greater coverage and fewer false negatives.

Frequent itemset mining based approach

The frequent itemset mining based approach uses the program's statements or structural pattern sequences to extract frequently occurring subsequences as specifications. For example, PR-Miner uses the frequent closed-itemset mining algorithm FPclose (Grahne and Zhu. 2003a) to mine the co-occurrences of statements in the function. This approach can mine the co-occurrence relationships among multiple APIs but does not reflect the semantic relationships between APIs (Li and Zhou 2005). In addition, the false positive rate in the mining results is high. In the evaluation of PR-Miner, 45 of the top 60 violations for the extracted rules in the violation report for PostgreSQL were false positives caused by false programming rules. In addition, when dealing with multiple API call sequences, the API sequence will be split into multiple pairwise sequences, such as [A, B, C] will be split into [A, B], [A, C], [B, C], which will lead to redundancy in subsequent rule extraction. Sequences without semantic relationship become the interference items of specification mining.

Henkel et al. proposed a specification mining approach based on unsupervised learning (Henkel et al. 1904). The approach assumes that APIs in a specification have similarities in the function name. Then it clusters the APIs with function names. The frequent itemset mining is conducted with the projection of elements in the domain of clusters.

PR-Miner uses FPclose algorithm to get all closed subitemsets labeling with different threshold as confidence for the corresponding rule. The algorithm overcomes the problem of traditional frequent pattern mining that generates excessive pattern results according to threshold settings.

There are several works that highly related to maximal pattern mining, Unil Yun et al. proposed more efficient maximal weighted frequent pattern mining considering weight information as well as the support values based

on tree and array structures((Lee and Yun 2018; Yun and Lee 2016; Yun et al. 2016)), they also use approximate weighted maximal frequent patterns considering error tolerance(Lee et al. 2016).

Template-based specification

The template-based specification mining approach uses a defined specification template to filter out the sequences that match its patterns and adopts a statistical approach to select a high-support pattern as a specification (Bian et al. 2018b). For example, Lemieux regards a program as a linear execution sequence of statements (Lemieux et al. 2015), extracts the proposition that satisfies the user-specified temporal logic template, and uses a statistical approach to mine the property with the highest confidence as the true proposition. Then, this method uses the true proposition to construct the temporal logic as a specification. Yun et al. (Yun et al. 2016) records the API node sequence and symbolic execution path by performing a lightweight static symbolic execution. Then, the frequency of context patterns, including the return value of a single API, the constrained causal relationships of the API pairs, and the implicit pre-and post-condition relationships of API pairs in the recorded result, is used to find the return processing of a single API and the control dependency relationship specification between pairs of APIs. Such approaches can mine specifications targeted by a particular specification template. However, the types of specifications that are mined are limited to the types defined in the template and the semantic relationships extracted are only between specific API pairs. A framework was proposed for API usage constraint and misuse classification which describe several typical templates (Schlichtig et al. 2022).

Template-based approaches are highly targeted and have a low false positive rate, but they cannot effectively mine relationships for specifications containing multiple APIs. If they are directly extended to mine multiple API specifications, the complexity of the algorithm increases rapidly according to the number of possible API combinations and number of potential semantic relationships, which can lead to scalability problems. For example, in (Chang et al. 2008, 2012), the frequent isomorphic graph minor of the program dependency graph centered on the selected candidate API is used as a specification. The specification mining problem is then modeled as a frequent graph minor mining problem, which is an NP-complete problem. In complex protocol handling programs, there are often call specifications between multiple APIs. Therefore, how to efficiently and accurately mine call specifications among multiple APIs is critical to the security analysis of these programs. In our approach, we use a two-stage process and focus on API sequence mining and semantic relationship extraction in different stages to improve scalability.

Document-based method

There are also some methods, which extract specifications from official or community documentation of open-source software for matching. Lv et al. (Lv et al. 2020) proposed method useing NLP to extract integration assumptions from the library documents and then verify the consistency with the APIs used in a program. Wang et al. (Wang and Zhao 2023) combined source code and documents to cross-validate and extract patterns.

Document-based methods extract relationships of a code fragment (function scope, file scope, et al.), most of them extract only return value of pairwise APIs, which has drawbacks on completeness of specification. In addition, the inaccuracy of descriptions of the documents may introduce deviation.

Problem definition

In this section, we further explain the problems we need to solve through motivating examples. The functions for tasks such as access control and protocol processing in programs are mainly implemented in the code base through API calls. The APIs are the main carrier for the interface and encapsulate the internal states of the program. Missing, out-of-order, and lacking checks for API calls can lead to security breaches and performance degradation. Specification mining extracts a set of associated APIs from multiple instances of API calls to verify their correct usage. In these instances, the necessary condition checks and calling context restrictions related to the associated API are an important part of the specification. The following typical examples of code introduce the goal of our approach.

Figure 1a shows the code snippet that implements time-stamp verification in the well-known cryptographic library OpenSSL, which calls certificate verification APIs to verify the certificate contained in the time-stamp signature. The following five APIs are called in the code snippet to implement the certificate verification:

1. X509_STORE_CTX_init, which initializes the certificate verification environment.
2. X509_STORE_CTX_set_purpose, which sets the certificate verification purpose.
3. X509_verify_cert, which verifies the certificate.
4. X509_STORE_CTX_get1_chain, which obtains the certificate chain information after the verification is successful.
5. X509_STORE_CTX_free, which cleans up the certificate verification environment.

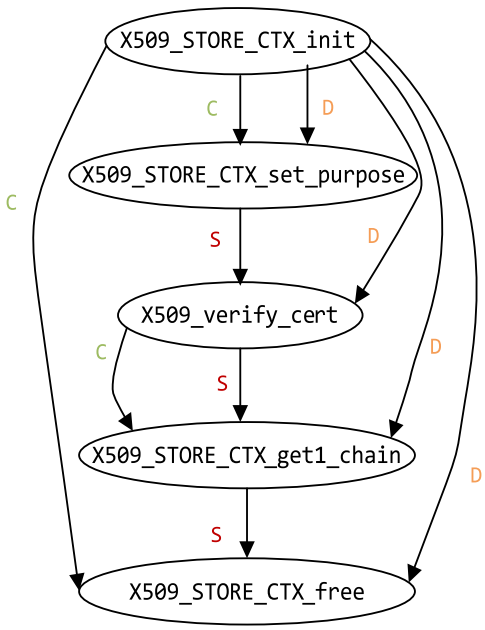
We found that violations of the following three situations will cause security problems.

1. The API calls cannot be missing

Figure 1b shows relationship of APIs mentioned above. In the sequence of API calls shown in Fig. 1a, the relevant APIs cannot be missing. The initialization operation performed by X509_STORE_CTX_init on line 258 is a prerequisite for all subsequent API calls. If the code does not call X509_STORE_CTX_set_purpose on line 260, the purpose of the verification will be ambiguous. If X509_STORE_CTX_free is not called on line 273, it will cause a memory leak in the certificate verification environment.

```
openssl-1.1.1\crypto\ts\ts_rsp_verify.c
168    X509_STORE_CTX *cert_ctx = NULL;
……
178    if (!X509_STORE_CTX_init(cert_ctx, store, signer, untrusted))
179       goto end;
180    X509_STORE_CTX_set_purpose(cert_ctx,
X509_PURPOSE_TIMESTAMP_SIGN);
181    i = X509_verify_cert(cert_ctx);
182    if (i <= 0) {
……
          }
189   *chain = X509_STORE_CTX_get1_chain(cert_ctx);
……
197    X509_STORE_CTX_free(cert_ctx);
```

(a)



(b)

Fig. 1 OpenSSL code snippet implementing the time-stamp protocol

2. The order of invocation and control dependencies among these APIs cannot be violated

A later API is called on the premise that an earlier API has already been called. In Fig. 1b, the edges labeled C indicate the control dependencies between the relevant APIs. For example, X509_STORE_CTX_init and X509_STORE_CTX_set_purpose must be called before X509_verify_cert. At the end of the certificate verification, X509_STORE_CTX_free must be called to clear the environment. In addition, as shown on lines 258 and 273, the initialization API X509_STORE_CTX_init and the end API X509_STORE_CTX_free appear as a pair, but a X509_STORE_CTX_init call does not necessarily follow a call of X509_STORE_CTX_free. When the X509_STORE_CTX_init call fails, the X509_STORE_CTX_free function cannot be called; otherwise, a NULL pointer reference problem will occur. X509_STORE_CTX_get1_chain must be called if X509_verify_cert returns a certificate verification success.

3. The semantic relationships (data dependence and parameter sharing) cannot be missing

Again, the data dependence and parameter sharing relationships are also an important part of the specification. In Fig. 1b, the edges marked with D indicate the data dependence relationships between the related APIs, and the edges marked with S indicate the shared parameter relationships among the related APIs.

For instance, X509_STORE_CTX_init creates the shared parameter relationships among the related APIs. For instance, X509_STORE_CTX_init creates the cert_ctx certificate validation environment and the rest of the APIs share the cert_ctx parameter. Data dependence and parameter sharing can distinguish whether the call sites of a series of APIs are semantically associated. Therefore, it is necessary to analyze them. In this paper, we refer to the above three relationships of APIs as semantic relationships.

Clearly, the usage specifications of these APIs should include not only whether they should appear together, but also the control and data flow relationships among them, expressed as a condition statement, related parameters, and the return values. Fig. 2 shows the client-side code of the SSL protocol in mbed TLS, which is ARM's open-source encryption library. If the code snippet does

```
mbedtls-development\programs\ssl\dtls_client.c
110  mbedtls_ssl_init( &ssl );
111  mbedtls_ssl_config_init( &conf );
……
183  if( ( ret = mbedtls_ssl_setup( &ssl, &conf ) ) != 0 )
     {
         goto exit;
     }
189  if((ret = mbedtls_ssl_set_hostname(&ssl, SERVER_NAME))!= 0 )
     {
         goto exit;
     }
209  do ret = mbedtls_ssl_handshake( &ssl );
     while( ret == MBEDTLS_ERR_SSL_WANT_READ ||
         ret == MBEDTLS_ERR_SSL_WANT_WRITE );
……
229  if((flags = mbedtls_ssl_get_verify_result(&ssl))!= 0)
     {
         mbedtls_printf( "%s\n", vrfy_buf );
     }
……
251  do ret = mbedtls_ssl_write( &ssl, (unsigned char *) MESSAGE, len );
     while( ret == MBEDTLS_ERR_SSL_WANT_READ ||
         ret == MBEDTLS_ERR_SSL_WANT_WRITE );
……
273  do ret = mbedtls_ssl_read( &ssl, buf, len );
     while( ret == MBEDTLS_ERR_SSL_WANT_READ ||
         ret == MBEDTLS_ERR_SSL_WANT_WRITE );
……
308  do ret = mbedtls_ssl_close_notify( &ssl );
     while( ret == MBEDTLS_ERR_SSL_WANT_WRITE );
     ret = 0;
331  mbedtls_ssl_free( &ssl );
```

Fig. 2 SSL API calls in main function of dtls_client.c in the mbed TLS library

not call the mbedtls_ssl_get_verify_result API in line 229 to verify the certificate or if mbedtls_ssl_read and mbedtls_ssl_write are called without evaluating their return values, the related device may construct a fake certificate to disguise itself as an SSL server, bypassing the verification process of the client program in the embedded system and enabling a man-in-the-middle attack. Further, if mbedtls_ssl_free is not called after mbedtls_ssl_read and mbedtls_ssl_write, the resource is not released. Finally, if the mbedtls_ssl_close_notify API call on line 308 does not conform to the specification, the connection will be unstable, and the performance of the communication will be heavily degraded.

In addition, inter-procedure analysis should be conducted in the specification mining process. The call site of the APIs in the sequence could be distributed over various functions in a function call chain, such as in the OpenSSL time-stamp response code in Fig. 3. Here, TS_RESP_new and TS_RESP_free should be called as a pair, but in TS_RESP_create_response, there is only a direct call to TS_RESP_new. The call to TS_RESP_free is executed indirectly through TS_RESP_CTX_free, and intra-procedure specification mining approaches cannot discover this association.

It will introduce important risks without data flow and parameter sharing analysis. As shown in Fig. 4, the BN_generate_dsa_nonce function in OpenSSL-1.1.1 is called by the ecdsa_sign_setup function to generate a random number k in the ecdsa signature to protect the weak random number generator. The k value requires strict protection to prevent leakage. The k value should be reset to zero after usage. A calling specification was shown in Fig. 13. As it is shown, the call sequence conforms to the specifications from the perspective of call sequence and completeness. However, the function OpenSSL_cleanse is not called to clean SHA512_CTX, but to clean variable private_bytes from the perspective of parameter sharing. The risk is that after k is generated and hash calculated by SHA512, the k value can be restored by the data left in memory of SHA512_CTX in a particular situation.

Similarly in pageant handle msg function in Pageant.c of Putty (shown Fig. 16). The hash execution environment data structure is not cleaned after MD5Final is called, which will lead to potential memory leaks.

In summary, the goal of specification mining is to mine the associated API sequences that must co-occur in the program and analyze the control dependencies and data flow relationships among the APIs in the sequence.

As described in Section Related work, current frequent itemset mining based approaches mine the co-occurring relationships of multiple APIs without considering their semantic relationships. They lack flow-sensitive, path-sensitive, and context-sensitive analysis and do not obtain information about the data and control flows of the APIs. In addition, these approaches

```
openssl-1.1.1\crypto\ts\ts_rsp_sign.c
378 TS_RESP *TS_RESP_create_response(TS_RESP_CTX *ctx, BIO *req_bio)
{
386   if ((ctx->response = TS_RESP_new()) == NULL) {
……
        goto end;
}
425   ts_RESP_CTX_cleanup(ctx);
}
438 static void ts_RESP_CTX_cleanup (TS_RESP_CTX *ctx)
{
……
442   TS_RESP_free(ctx->response);
……
}
```

Fig. 3 TS_RESP_create_response function in ts_rsp_sign.c of OpenSSL-1.1.1 and its time-stamp API calls

```
openssl-1.1.1a\crypto\bn\bn_rand.c
205     int BN_generate_dsa_nonce(BIGNUM *out, const BIGNUM *range,
                          const BIGNUM *priv, const unsigned char *message,
                          size_t message_len, BN_CTX *ctx)
{
209    SHA512_CTX sha;
……
223    k_bytes = OPENSSL_malloc(num_k_bytes);
224    if (k_bytes == NULL)
               goto err;
….
241    for (done = 0; done < num_k_bytes;) {
242        if (RAND_priv_bytes(random_bytes, sizeof(random_bytes)) != 1)
243            goto err;
244        SHA512_Init(&sha);
245        SHA512_Update(&sha, &done, sizeof(done));
246        SHA512_Update(&sha, private_bytes, sizeof(private_bytes));
247        SHA512_Update(&sha, message, message_len);
248        SHA512_Update(&sha, random_bytes, sizeof(random_bytes));
249        SHA512_Final(digest, &sha);
}
……
266  err:   OPENSSL_cleanse(private_bytes, sizeof(private_bytes));
267       return ret;
}
```

Fig. 4 The code of BN generates dsa nonce function in OpenSSL-1.1.1 calls hash related API

lack inter-procedure analysis. The security-sensitive function-based approaches are limited to the semantic relationship analysis of the functions that are specified as sensitive. Moreover, in template-based and security-sensitive function-based approaches, the target program is represented in the form of symbolic paths or program dependency graphs. The data flow and control flow information are included in the mined specification. However, these approaches are difficult to extend to tasks that mine specifications for multiple APIs.

To address this problem, we propose a two-stage API call specification mining approach that that efficiently extracts a relatively complete list of the API combinations and semantic relationships between APIs. It focuses on API sequence mining and semantic relationship extraction in different stages to mine specifications for multiple APIs in a scalable way.

Semantic relationship sensitive API specification mining approach

Current popular open-source projects usually have strong modularity with good API design, and in the main project code, the relevant APIs are correctly called in a certain number of instances. Under these assumptions, this study focuses on specification mining for open-source projects. Our approach efficiently extracts a relatively complete list of the API combinations for the code of the target program under analysis, which reduce the unnecessary analysis when dealing with multiple APIs in co-occurrence relationship. Based on this, the semantic relationship between APIs is extracted for an accurate analysis with lower costs compared with traditional symbolic execution. The main process of this approach is shown Fig. 5. which consist of two stages. The input of

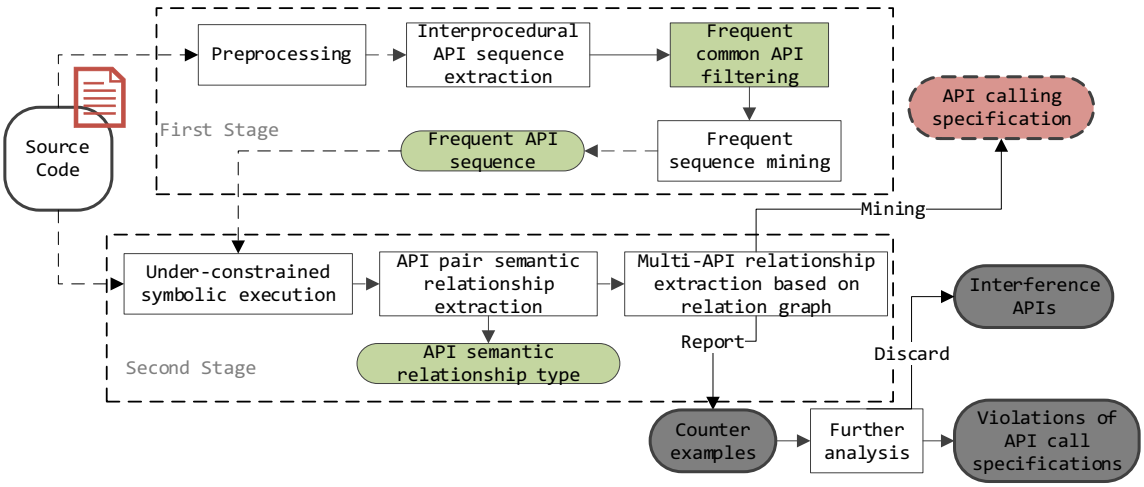


Fig. 5 Overview of the proposed specification mining approach

our approach is the source code of the projects, and the API calling specification is mined as output. In the process, counter examples will be reported and analyzed to distinguish interference APIs and violations of API call specifications.

Stage 1: Frequent API set mining based on frequent common API identification and filtration.

In this stage, frequent API set mining is performed on the given open-source project to find frequent API sequences. First, the number of API calls related to the specifications may be small in some projects, which makes it difficult to extract valid API sequences through frequent itemset mining approaches. Hence, we use client code calling these libraries for specification mining. Then, a global interprocedural analysis is performed on the target source code, and a call graph is constructed to extract inter-procedure API sequences by traversing each node (which corresponds to a function) in the call graph. Then, for the extracted sequence set, the maximal frequent itemset mining approach (Grahne and Zhu 2003a, 2003b) is used to obtain frequently appearing API sequences. Some APIs in the program have a higher overall frequency and are distributed among multiple frequent sequences, but the correlation between these APIs and the APIs that implement the main function is not strong. Typical examples include APIs related to error handling and log operations. This increases the number of frequent APIs that are not semantically related, affecting the quality of the specification and occupying a large amount of algorithm execution time. Hence, such APIs are first discarded before frequent API mining is performed. This effectively improves the quality of the specification and the efficiency of the algorithm.

Stage 2: API semantic relationship specification mining based on domain adapted under-constrained symbolic execution and graph based relationship aggregation.

This stage performs a domain adapted under-constrained symbolic execution to extract the API semantic relationship specification from frequent API sequences of the first stage. First, a path-and flow-sensitive analysis of the code of the target program under analysis is performed for the under-constrained symbolic execution, and the frequent API sequences are used as trigger points to record the symbolic information and path constraint information of the API in these sequences. Then, the control dependencies among the APIs are extracted from the path constraint information, and the pairwise relationships of shared parameters and return value data dependencies for the APIs are extracted from the symbolic information of the API call site. Using the API sequences and their pairwise semantic relationships, the API relationship graph is constructed, and the relationships are aggregated. The context-sensitive, path-sensitive, and flow-sensitive specifications for multiple APIs are mined from the graph. The scope of related symbolic path and constraint information collection at this stage is limited to the relatively small-scale API sequence set in the frequent sequences. Moreover, only the relationships of the APIs in this limited set are mined, which significantly reduces the scope and the search states of the relationships.

Frequent API set mining based on frequent common API identification and filtration

Definition of frequent common API

We define frequent common APIs illustrated below as interference APIs, which interfere the mining frequent

APIs. In the remainder of this section, we would explain the motivation.

In the first stage of analysis, the API sequences are first extracted from the code of the target program under analysis, and the frequent itemset mining algorithm is used to mine API sequences with two or more combined APIs to find the API co-occurrence relationships. A frequent itemset is a set of sequences that frequently appear in a data set and whose degree of support is greater than or equal to the minimum support degree (min_sup), where the support degree refers to the frequency at which a certain set appears in all sequences. For specification mining, the existing frequent itemset-based approach has the following three problems.

1. For some target open-source projects, specification-related APIs may occur infrequently, resulting in insufficient support. This makes it difficult to extract valid API sequences.
2. Without context-sensitive analysis, it is difficult to find inter-procedure API sequences and related specifications.
3. There is no filtration of frequently occurring common functions. If these APIs are not filtered, they are likely to form frequent co-occurring relationships with other APIs, resulting in the output of many redundant or unrelated API sequences, increasing the number of invalid specifications and using up the algorithm's calculation resources. Therefore, frequent common APIs should be filtered to reduce interference.

To address the first problem, the following two strategies are adopted to increase the frequency of the specification-related APIs: 1) related code is added by introducing other project files that reference the code base and 2) functional verification and performance testing code that large open-source projects usually provide (usually in the "test" folder) is added. These additional lines of code usually do not consider security, which is reflected in their non-compliance with the specified semantic relationships among APIs. In the second phase of the analysis, we exclude these supplementary lines of code.

For the second problem, a global inter-procedural analysis is performed to extract the API call sequences across functions by means of a function call graph constructed by the compiler framework. First, the source code is analyzed by the compiler front end, and the abstract syntax tree and control flow graph in the function are constructed through lexical and syntax analysis. Then, using the call relationships between functions, a global function call graph is defined. In the function call graph, a node represents a function, and an edge represents a calling relationship between the functions. Then, using the source code compilation engine, each node in the function call graph is traversed and the inter-procedure API call sequence is extracted.

For the third problem, we combine the filtration of interference APIs with the classic FPMAX algorithm to mine maximally frequent API sequences to improve the quality and efficiency of specification mining (Grahne and Zhu 2003b).

The approach of PR-Miner use FPclose algorithm to mine only the closed sub-itemsets. A closed sub-itemset is the sub-itemset whose support is different from that of its super-itemsets (Chang and Podgurski 2012). For example, in an itemset database

$$\mathrm{D} = \{\{\mathrm{a,b,c,d,e}\},\{\mathrm{a,b,d,e,f}\},\{\mathrm{a,b,d,g}\},\{\mathrm{a,c,g,h}\}\}$$

The frequent sub-itemset {b}, {d}, {a,b}, {a,d} and {b,d} are not closed since their supports are the same as their super itemset {a,b,d}$\{a,b,d\}$. FPclose generates the closed sub-itemsets {a}$\{a\}$:4 and {a,b,d}$\{a,b,d\}$:3 as closed sub-itemset. While FPMAX would generates the itemsets {a,b,d}$\{a,b,d\}$:3 as maximal frequent sub-itemset with the minimum support threshold of 3. In our approach, we only mine the frequent API sequence exceeded the specified threshold. One of our goals is to get a relatively complete API calling sequence including all the subsequences without considering the different confidence of the subsequences. So, we use FPMAX instead of FPclose and FPGrowth algorithm.

The classic FPMAX algorithm requires each element to appear only once in each call sequence, so we deleted duplicate APIs in the sequences. Then, the total frequency of each API in the sequences is counted, and those APIs whose frequencies are lower than the minimum support are filtered out of each sequence. Then, the APIs are sorted in descending order according to their frequency. The APIs in each sequence and their frequencies are inserted into the FP tree one by one to construct it. The insertions begin at the root node. If the API to be inserted does not exist inthe tree, a new branch is created. Otherwise, the frequency of the API is added to the frequency of the corresponding node. Fig. 6 shows a portion of an FP tree constructed from the API sequences extracted from the libTIFF-4.0.10 code. Here, ϕ is the root node of the FP tree, and the remaining node show the name of the corresponding APIs and their frequencies on the path. Each vertical path (solid line connection) in the FP tree is a data item set that satisfies the minimum support degree in the sequence. To quickly

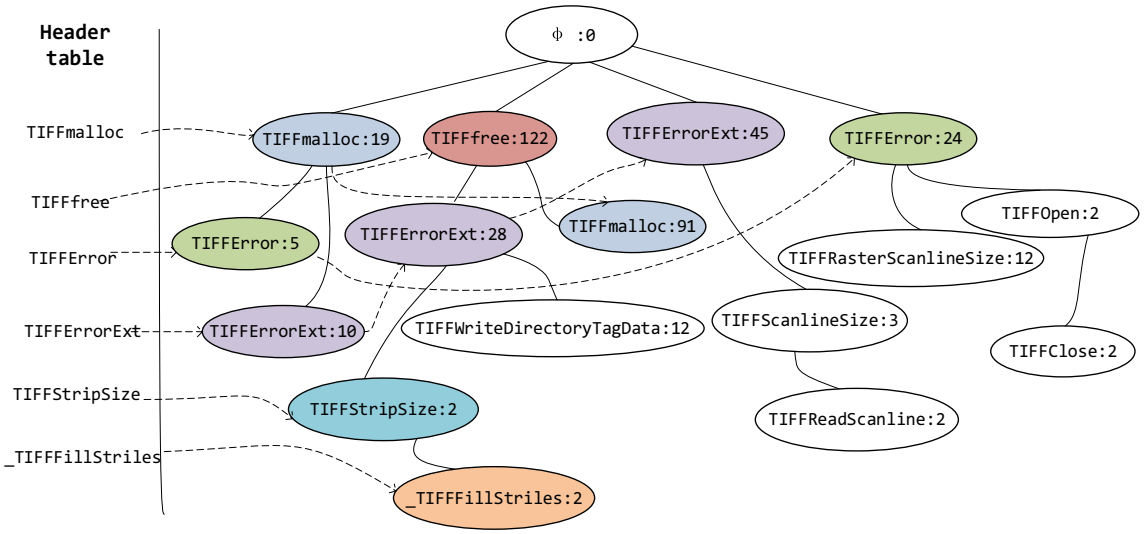


Fig. 6 Part of an FP tree for the API sequences mined from libTIFF-4.0.10

access the same items in the tree, all the same items are connected using a linked list through the header table, which points to the linked list and is represented in the Fig. 6 by horizontal dashed lines.

By analyzing the characteristics of the nodes of interference APIs (TIFFErrorExt and TIFFError) in the FP tree Fig. 6, we can find that a candidate interference API has the following three characteristics:

1. The overall frequency is over a certain value. According to its definition, an interference API must frequently appear in the code of the target program under analysis.
2. The node appears in the upper layers and is closer to the root node. Because the overall frequency of interference APIs tends to be higher, when the elements of the API sequence are sorted by frequency, they appear at the top. Hence, when the FP tree is constructed in this order, they are always in the upper layers of the tree.
3. Its out degree exceeds the average value. Interference APIs appear in multiple API sequences.

When these sequences are inserted into the FP tree, the insertion traversal usually first passes through nodesrepresenting frequent common APIs, and then the node is inserted below nodes corresponding to APIs with a relatively low frequency. Interference API nodes act as the parent nodes of multiple less frequent API nodes and usually have more branches.

Improved FPMAX algorithm

In the improved FPMAX algorithm, the maximal frequent API sequence is tracked by the global MFI tree (Maximal Frequent Itemset tree). The MFI tree also starts from root node ϕ. For each frequent sequence extracted through the FP tree, if the sequence is not a subsequence starting at any node in the MFI tree, it is inserted into the MFI tree. The insertion of sequences into the MFI tree is the same as the insertion of nodes into the FP tree. The algorithm starts with a sequence initialized to be empty as a prefix of the FP tree, and performs recursive processing as follows:

1. If there is only one path in the tree, set the level of path support to the minimal value of frequency of each API. If the support level is greater than the minimum support level, insert the path into the MFI tree and return.
2. Identify and filter out interference APIs from the child nodes of the root node according to the frequency and degree of the nodes. Put the interference APIs into the look-aside list and reconstruct the FP tree after filtration.
3. Extract each node pointed to by the header table in the FP tree as well as its frequency. Link them to the prefix sequence to form a new prefix. The support level of the new prefix is the minimum value of the frequency of each API.
4. If the support is less than the minimum support, discard the prefix.

```
Procedure FPMaxR(T,minsup)
Input: T: FP-TREE
Global: MFIT: the MFI-tree LookAside:List
Output: The MFIT that contains all MFI. LookAside: Frequent common API
1.  if T only contains a single path P
2.   if MFIT doesn't contains T.base∪P
3.    insert T.base∪P into MFIT
4.    Return;
5.  for each child A of T.ROOT do
6.    if A.count>midnum and count of a.outedges >midout
7.       put A into LookAside
8.       put A off T,Adjust T
9.  for each i in T.header do
10.   Y=T.base∪{i} with i.count
11.   construct Y's conditional FP-tree T_y with min_support of minsup
12.   if T_y ≠ ∅
13.     Call FPMaxR (T_y,minsup);
```

Fig. 7 Pseudocode for the proposed FPMAX algorithm with frequent common API processing

5. If the support is greater than the minimum support, construct a conditional FP tree that removes the node, and call the algorithm recursively with the new prefix and the conditional FP tree as parameters.

Figure 7 shows the pseudo code of the improved FPMAX algorithm. Here, input T represents a constructed FP tree containing three fields: base, header, and root. "T.base" is the prefix of the current tree to be mined, "header" represents the header table of the FP tree, "LookAside" is a candidate list for interference APIs, and "MFIT" is a global MFI tree structure. The description of the algorithm Fig. 7 is based on the original FPMAX algorithm inGrahne and Zhu 2003b by supplementing the processing of interference APIs in Lines 5–8. Lines 1–4 of the algorithm are the terminal path of the recursive process. When there is only one path in the tree and the support is greater than the minimum support, the path and its support are inserted into the MFI tree. Lines 5–8 filter the interference APIs and reconstruct the filtered FP tree. In line 6, "midnum" is the minimum support number of interference APIs and "midout" is the lowest out degree of interference APIs. The values of midnum and midout are calculated by finding the median of support number and out degree of the child nodes of the root of the initial FP tree. Lines 9–11 build and recursively analyze the conditional FP tree (represented by T_y T) for each node. In lines 12–13, when the conditional FP tree is not empty, the algorithm is called recursively with the conditional FP tree as input.

After this process, the candidate interference APIs in the LookAside list were distinguished. Frequent common APIs, such as {CRYPTO_malloc, CRYPTO_free} in OpenSSL and {_TIFFmalloc, _TIFFfree} in the libTIFF library, have obtained plenty of co-occurrence relationships and less significance for subsequent semantic analysis. Other functions like TIFFError and TIFFErrorEXT in the libTIFF library, which obtain a high frequency in calling sequence, also increase the cost of analysis.

API semantic relationship specification mining based on domain adapted under-constrained symbolic execution and graph-based relationship aggregation

The frequent API sequences mined in Sect. "Semantic relationship sensitive API specification mining approach" are frequently occurring API combinations that may be semantically independent of each other. The second phase further extracts control and data dependencies on the parameters and return values of

the APIs in the frequent sequences of the code of the target program under analysis so that flow-and path-sensitive specifications may be mined. The current flow-and path-sensitive API call specification mining approaches have the following two problems:

1. The analysis, extraction, recording, and matching of all APIs in the code incur large computational and storage overheads with low scalability.
2. There is no semantic relationship matching among multiple APIs. When searching for dependence relationships between arbitrary APIs, as the number of APIs increases, the number of possible relationships between APIs increases rapidly. As a result, the search runtime increases sharply, reducing scalability.

In this section, we present our method for extracting the symbolic path constraint information for APIs in the frequent sequences output by the first stage and construct the API relationship graph according to the relationships between pairs of APIs to mine multiple API call specifications.

Domain adapted symbolic path constraint information extraction

For the first problem, we propose an under-constrained symbolic execution path information extraction approach. The scope of the symbolic execution path recording, and constraint information analysis are limited to the APIs in the frequent sequences. As shown Fig. 8, the variables involved are first symbolized and updated by traversing the statements of all the paths of the program from the entry point of the code control flow graph. Then, the statements in the program are parsed into expressions with symbolic variables and constants. If a control statement is encountered, a path-sensitive analysis is performed, and both paths of the branch targets are independently explored while appending the control condition to the related path constraint. When traversing any of the APIs in the frequent API sequence set, the path constraint is solved and the symbol execution environment information, which includes the name of the API node, the symbol execution path constraint, the corresponding parameters, and return value information, is recorded. Finally, the analysis results are recorded. By reducing the range of symbol path information that is

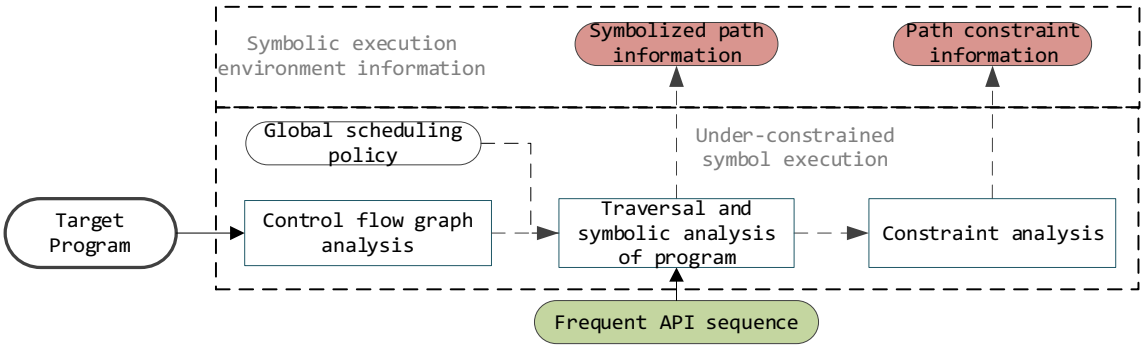


Fig. 8 Symbol execution analysis based on frequent API sequence

Table 1 Symbol types for the symbol execution trace

Type	Name	Symbolic form
API	*function*	$f_x, \left(arg_x(1), arg_x(2) \ldots arg_x(m)\right), (ret_x(1), ret_x(2) \ldots ret_x(n))$
		$f_x, \left(arg_x(1), arg_x(2) \ldots arg_x(m)\right), (ret_x(1), ret_x(2) \ldots ret_x(n))$
Symbolic variable	*var*	$arg_x(i) i \in \{1, m\} \mid ret_x(j) j \in \{1, n\}$ arg$_x$(i) i ∈ {1, m}|ret$_x$(j) j ∈ {1, n}
Constant	*const*	*num*|*srtringnum*|*string*
Comparison operator	*Δcmp*	= |≠| < |>| ≤ | ≥=|≠|<|>≤|≥
Expression	*exp*	$var_1 \Delta cmpvar_2 \mid var \Delta cmpconstvar_1$ Δ*cmp*var$_2$|var*Δcmp*const
Logical	*ΔL*	|∧|∨
Constraint information	*constraint*	$exp_1 \Delta Lexp_2$
API call	*functionCall*	*function,constaint*
Trace	*functionCallSeq functionCallSeq*	*functionCall*+

recorded and reducing the number of instances of constraint solving, this approach greatly reduces the computing and storage overheads. It also reduces the scope of the analysis in the next stage.

Table 1 lists the definitions of the symbols recorded during symbolic execution. Here, "trace" represents the recorded symbolic path information, which included the sequence of the API in the path, and each record of API information includes the name, parameter, return value, and path constraint information.

Multiple API call specification mining

To address the second problem, this section presents an API relationship graph that combines pairwise semantic relationships of APIs to obtain a multi-API call specification. We first define the types of pairwise semantic relationships of APIs. Then, using the symbolized path information previously obtained, the relationships are matched to pairs of APIs. Then, using these matches, the API relationship graph is constructed, and the multi-API call specification is obtained by searching the connected subgraph. Because the symbolic path information only includes APIs that appear in frequent API sequences and the number of possible relationships between a limited number of API pairs is relatively small, the computational overhead of the proposed multi-API call specification mining is greatly reduced.

1. Types of API pairwise semantic relationships and their matching

The analysis in Section Related Work demonstrates that the semantic relationship between API pairs usually includes control dependencies, data dependencies, and parameter sharing relationships. The control dependency relationship of an API pair is reflected by the fact that the return value of an API helps to determine whether to execute another API. For example, the mbed TLS APIs mbedtls_ssl_read/mbedtls_ssl_write should be called only after the mbedtls_ssl_get_verify_result call returns successfully, so their path constraint contains the return value of mbedtls_ssl_get_verify_result (shown Fig. 2). A data dependency occurs when the return value or vari-ables defined by one API and transferred through parameters to another API. For example, the mbed TLS APIs mbedtls_ssl_init is used as the parameter of mbedtls_ssl_handshake (shown Fig. 2). Parameter sharing occurs when a pair of APIs have shared parameters. For example, mbed TLS APIs of mbedtls_ssl_close_notify and mbedtls_ssl_read have a common "&ssl" parameter (shown in Fig. 2).

The matching rules for determining whether an API pair $(f_x f_y)$ has the above three relationships are shown in Fig. 2. Here, $\text{arg}_x(\text{i})$ is $\text{arg}_y(\text{j})$ means that parameter $\text{arg}_x(\text{i})$ in function f_x is the same variable as $\text{arg}_y(\text{j})$ parameter in function f_y, and constraint(f_y) $constraint(f_y)$ denotes the symbolic variable contained in the path constraint information of function f_y f_y.

When mining the relationship between API pairs, we use a corresponding relationship support matrix for tracking the support of the relationship for each API pair

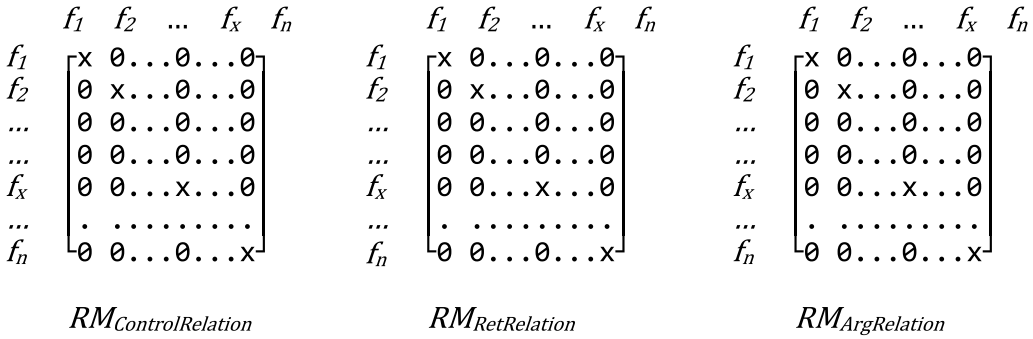


Fig. 9 Relationship support matrix

Table 2 API relationship matching rules

Relation type	Matching rule
Control dependence	$ControlRelations(seq) = \{(f_x, f_y) \mid \exists i, ret_x(i) \in \text{constraint}(f_y)\}$ $ControlRelations(seq) = \{(f_x, f_y) \mid \exists \text{i}, \text{ret}_x(\text{i}) \in constraint(f_y)\}$
Data dependence	$RetRelations(seq) = \{(f_x, f_y) \mid \exists i, j, ret_x(i) is arg_y(j)\}$ $RetRelations(seq) = \{(f_x, f_y) \mid \exists \text{i}, \text{j}, \text{ret}_x(\text{i}) \text{is } \text{arg}_y(\text{j})\}$
Parameter sharing	$ArgRelations(seq) = \{(f_x, f_y) \mid \exists i, j, arg_x(i) is arg_y(j)\}$ $ArgRelations(seq) = \{(f_x, f_y) \mid \exists \text{i}, \text{j}, \text{arg(i)} \text{is } \text{arg}_y(\text{j})\}$

Table 3 Relationship type tags and corresponding dependencies or shared variable set representations

Relation type	*L*	τ
Control dependence	C	*NULL NULL*
Data dependence	D	$RET_x \cap ARG_y$, $RET_x \cap ARG_y$, RET_x represents the return variables set of f_x and parameter variables passed to f_x which is also defined in the function f_x. ARG_y represents the parameter variables set of f_y
Parameter sharing	S	$ARG_x \cap ARG_y$, $ARG_x \cap ARG_y$, ARG_x represents the parameter variables set of f_x, ARG_y represents the parameter variables set of f_y

in the frequent API sequences for each relationship. The elements of this matrix are initialized to 0, as shown in Fig. 9, where $RM_{ControlRelation}$, $RM_{RetRelation}$, and $RM_{ArgRelation}$ respectively represent the support matrices of control dependence, data dependence, and parameter sharing relationships.

Then, for any function pair (f_x,f_y) (f_x,f_y) in each recorded symbolic path, the different relationships are matched according to the rules of Table 2. That is, if the symbolic variable of a return value of f_x is included in the symbolic variable set related to the path constraint of f_y, there is a control dependency relationship between f_x and f_y f_x,f_y. If the symbolic variable of a return value of f_x and the symbolic variable of a parameter in the parameter list of $f_y f_y$ are the same, then there is a data dependency relationship between f_x and f_y. If the symbolic variable of a parameter of f_x is the same as the symbolic variable of a parameter of f_y, then there is a shared parameter relationship between f_x and f_y corresponding element of (f_x,f_y) (f_x,f_y) in the relationship support matrix is increased by one. Finally, if the If there is a relationship between f_x and f_y, the corresponding element value of the function pair in a relationship support matrix is greater than a given threshold, the quad $(f_x,f_y,\langle L,\tau\rangle)(f_x,f_y,(L,\tau))$ is added to the corresponding relationship list, where the L represents the relationship type and τ represents the dependent or shared variable set. The representations of L and τ for different relationship types is shown Table 3.

2. Multi-API semantic relationship aggregation based on the API relationship graph

The paired API relationships are combined to obtain the usage specifications of multiple APIs. First, the frequent API sequences and the paired relationship list are used to construct an API relationship graph. Then, the algorithm to find the maximally connected subgraph of a non-connected graph (Karp and Tarjan 1980) proposed by Tarjian is used to find the largest connected subgraph in the API relationship graph. Finally, connected subgraphs containing at least two nodes are retained and the specifications for multi-API relationships are constructed according to the subgraphs.

For example, using the set of frequent API sequences {BN_CTX_new, BN_CTX_free, BN_CTX_start, BN_CTX_end, BN_CTX_get, BN_new, BN_free, BN_copy} obtained by analyzing the OpenSSL source code, we can construct the API relationship graph shown in Fig. 10. The nodes in the figure represent the APIs in the frequent API sequences. The three relationship lists are traversed, and an undirected edge is added between the nodes

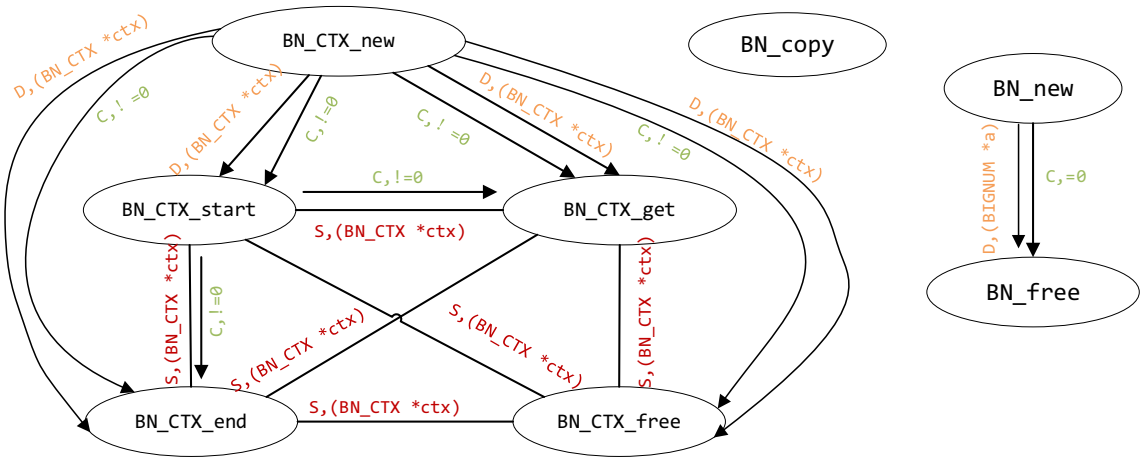


Fig. 10 API-pair relationship graph of a frequent API sequence in OpenSSL source code

```
Specification 1
APISequence:{ BN_CTX_new, BN_CTX_free, BN_CTX_start, BN_CTX_end, BN_CTX_get}
RelationSequence:
{ [BN_CTX_new,BN_CTX_free,<C,!=0>],
[BN_CTX_new,BN_CTX_start,<C,!=0>],
[BN_CTX_new,BN_CTX_end,<C,!=0>],
[BN_CTX_new,BN_CTX_get,<C,!=0>],
[BN_CTX_new,BN_CTX_free,<D,(BN_CTX *ctx)>],
[BN_CTX_new,BN_CTX_start,<D,(BN_CTX *ctx)>],
[BN_CTX_new,BN_CTX_end,<D,(BN_CTX *ctx)>],
[BN_CTX_new,BN_CTX_start,<D,(BN_CTX *ctx)>],
[BN_CTX_start,BN_CTX_end, BN_CTX_get, BN_CTX_free,<S,(BN_CTX *ctx)>],
[BN_CTX_new,BN_CTX_start,<M,NULL>],
[BN_CTX_start,BN_CTX_get,<M,NULL>],
[BN_CTX_get,BN_CTX_end,<M,NULL>],
[BN_CTX_free,BN_CTX_end,<P,NULL>],
[BN_CTX_free,BN_CTX_start,<P,NULL>],
[BN_CTX_end,BN_CTX_get,<P,NULL>],
}
Specification 2
APISequence:{BN_new,BN_free}
RelationSequence:
{[BN_new,BN_free,<C,!=0>],
[BN_new,BN_free,<D,(BIGNUM *a)>],
[BN_new,BN_free,<M,NULL>]
}
```

Fig. 11 Specifications of a frequent API sequence in OpenSSL source code

corresponding to the APIs for which a pair relationship exists. The labels of the edges represent different relationship types and dependent or shared variable sets. An API pair can have more than one semantic relationship. For instance, BN_CTX_new and BN_CTX_free have both control dependence and data dependence relationships. Hence, a node pair in the API relationship graph can have more than one edge. Actually, an API pair can have both control dependence and data dependence or both control dependence and a parameter sharing relationship. Three connected subgraphs can be found from the API relationship graph. The connected subgraph with the single node BN_copy is discarded. For each of the remaining two connected subgraphs, the API set and theAPI-pair relationship set in the subgraph are extracted to obtain the two specifications shown in Fig. 11.

Implementation and evaluation

Evaluation setup

Using the Clang static analyzer (from the LLVM framework), a specification mining tool called specificsan was implemented. The Clang static analyzer's symbolic execution engine performs context-and path-sensitive interprocess flow analysis for C/C++code. The graph reachability engine and the symbolic execution engine, which constitute the infrastructure for the data flow analysis, were used for analysis (Shastry et al. 2016).

In the first phase of the implementation, the API sequence extractor was implemented using the graph reachability engine, which tracks the call-return semantics (context sensitivity) of the procedure call and extracts the inter-procedure API sequence.

API sequences does it solve the path constraint and collect symbolic path and constraint information. In the analysis, the loop is unrolled only once, so that the path condition of each API obtained by the local analysis can be saved on one node and the analysis result can be reused in different contexts, thereby further improving the analysis efficiency. When matching the three types of semantic relationships between API pairs, the scope of the analysis is relatively small because the frequent API sequences have been extracted. To avoid missing relationships between APIs as much as possible and to exclude the occasional relationship that has a single instance, when the three types of semantic relationships between the API pairs are evaluated using the

Table 4 Test target open-source project information

Project name	Amount of code (KB)	Client code	Client code example
libTIFF-4.0.10	2649	8 debian packages using libTIFF	Ghostscript, libfox, libwraster, vagrant, etc
Openssh-7.9	3717	16 debian packages using Openssh	Vagrant,sshuttle,ssh-krb5,etc
mbedTLS-2.16.0	4206	Microchip's open-source test case code	Microchip's open-source test case code
OpenSSL-1.1.1	22,922	45 debian packages using Openssl	Tinyca openvpn dsniff ssvn, etc
Putty-3.4	2386	5 open-source projects using putty	PuttyRider, WebPutty, etc
FFmpeg-3.0.12	10,854	8 open-source projects using ffmpeg	FFMpegCore, mobile-ffmpeg, etc
MuPDF-1.14.0	54,272	7 open-source projects using MuPDF	Android-MuPDF, mppdf-qt, go-fiz, etc
php-7.0.22	19,148	5 open-source projects using php	PHPOfficeSpreadsheet, etc
Pidgin-2.13.0	13,209	5 open-source projects using Pidgin	Skyp4pidgin, pidgin-lwqq, etc
zlib-1.2.11	1464	8 open-source projects using zlib	Zlib-ng, zlib-searcher, etc

relationship support matrix, the threshold of support is set to two. The specification mining tests were carried out on several types of open-source projects as detailed in Table 4. Further experiments were implemented on six projects of them. libTIFF-4.0.10, OpenSSH-7.9, mbed TLS-2.16.0, OpenSSL-1.1.1, Putty-0.7 and zlib-1.2.11. Of these, OpenSSL and mbed TLS are well-known projects implemented with encryption and the SSL/TLS protocol. OpenSSH is a well-known encryption library project with SSH protocol. libTIFF is well-known open-source projects that deal with the complex formats of libTIFF. These projects are widely used, their API interfaces are rich, and the correctness of their API calls are important for security. Because of the low frequency of calls to the main function APIs in mbed TLS, Microchip's open-source test case code (MicrochipTech 2019), which references mbed TLS, was added. For libTIFF-4.0.10, Openssh-7.9 and OpenSSL-1.1.1, client code of debian packages using these libraries were used for specification mining.

Selection of the minimum support number of frequent item

During the mining process, we found that the different settings of the minimum support of frequent items have a large impact on the mining results of frequent API sequences. This section compares the mining results of each open-source project with different values for the minimum support of frequent items, analyzes the changes in mining performance with different settings, and selects the appropriate minimum support number. Fig. 12 shows the number of total frequent API sequences extracted from OpenSSL and Openssh as well as the effective sequences confirmed.

In the review process, we directly used the semantic relationship extraction method in the second step

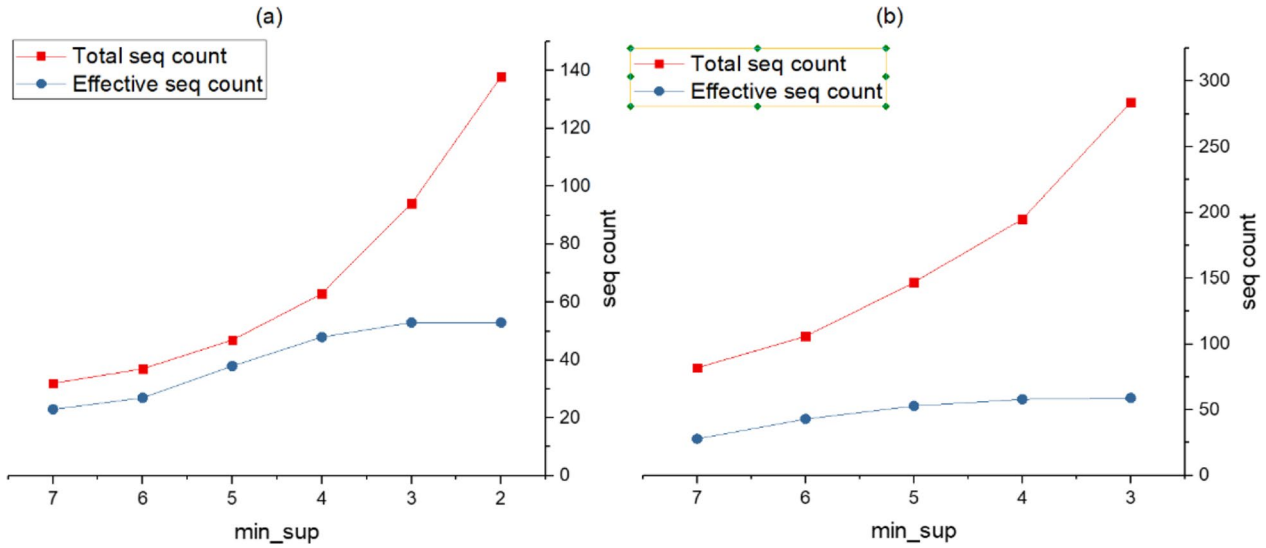


Fig. 12 Specification count and effective number for different minimum support values the FPMAX algorithm for **a** OpenSSL and **b** Openssh

of our approach. The judgment criteria were that if the items in the sequence do not have any semantic relationships with each other, we consider the sequence to be ineffective.

The minimum support number was dynamically and optimally selected for different target projects. As Fig. 12a shows, for OpenSSL, as the minimum support number decreases, the total number and effective number of mined frequent sequences increases. When the minimum support setting is reduced from 4 to 3, the total number is further increased, but the effective number does not increase, Hence, for OpenSSL, the minimum support number was set to 4. As Fig. 12b shows, for Openssh, when the minimum support number decreases from 5 to 4, the effective number no longer increases, so the minimum support number was set to 5. Using a similar experimental analysis for libTIFF, and mbed TLS, the minimum support numbers were chosen to be 3 and 4, respectively. It is worth mentioning that the minimum support number may be different due to the specific features of the open-source projects, the amount of code, etc. The range of minimum support numbers is limited. The experiments are conducted by estimation and prior analysis, and it's easy to find the optimal value.

Specification mining results

API frequent sequence mining results

Existing approaches such as PR-Miner and ml4spec only mine frequent API sequences, so this section compares the results of the PR-Miner approach with the results of the first phase of our approach. To test the effect of eliminating interference APIs on the mining results, the results of the first stage of the approach with and without filtering out the interference APIs are compared.

The effective sequences mined by the PR-Miner approach and those mined in the first stage of our approach are merged as the benchmark data of the frequent API sequences for each open-source project. Then, the recall and effective ratios of the different approaches are analyzed. The recall ratio

$$RR = TP/(TP + FN)$$

is the ratio of the correctly reported samples to all benchmark samples. The effective ratio

$$ER = TP/(TP + FP)$$

is the ratio of the correctly reported samples to the total number reported.

The results are summarized in Table 5. Compared with PR-Miner, the RR of the frequent API sequences in the first stage of our approach is much higher. This is because the PR-Miner approach extracts the API sequences intra-procedurally, and it may miss useful APIs in the sequence. Moreover, eliminating interference APIs has no effect on the RR of the first stage of the approach, but its ER of the first phase is significantly improved. This is because there are many interference APIs in these test objects (Table 6 shows the frequency of some interference APIs), and these interference APIs generate many invalid frequent sequences that contain them. By filtering

Table 5 Comparison of PR-Miner,ml4spec and the proposed method (first stage) results. FCA: frequent common APIs

Project	PR-Miner		ml4spec		FCA count	Without FCA removal		With FCA removal	
	RR(%)	ER(%)	RR(%)	ER(%)		RR(%)	ER(%)	RR(%)	ER(%)
libTIFF-4.0.10	60	55	66	84	7	74	63	74	85
OpenSSL-1.1.1	53	54	55	80	17	68	58	68	81
OpenSSH-7.9p1	56	64	62	78	11	72	74	72	82
Mbed TLS-2.16.0	57	63	61	74	15	73	63	73	77
Putty-0.7	58	62	63	77	13	70	62	72	78
zlib-1.2.11	60	64	60	76	10	66	70	74	80

Table 6 Number of frequent common APIs found in the experimental projects. FCA name: name of frequent common API, FR: frequency

Mbed TLS		OpenSSL		Libtiff		OpenSSH		Putty		Zlib	
FCA name	FR	FCA name	FR	FCA name	FR	FCA name	FR	FCA name	FR	FCA name	FR
Test_fail	302	ERR_PUT_error	3,425	TIFFErrorExt	591	ssh_err	750	safefree	384	gz_error	12
Mbedtls_platform_zeroize	16	BIO_printf	2,084	TIFFError	488	strerror	607	safemalloc	260	free	12
Mbedtls_debug_print_msg	55	ERR_print_errors	472	_TIFFfree	440	logit	273	saferealloc	91	_tr_flush_bits	7

them out, the invalid sequences are eliminated. At the same time, the elimination of interference APIs reduces the size of the data set and significantly reduces the processing time of the algorithm.

The RR of the proposed approach is significantly higher than the ml4spec approach. After analyzing the experimental data, it was found that the ml4spec approach used the text similarity method to cluster and filter the API based on the clustering process, and some textually dissimilar APIs were omitted. For example, in the valid sequence in OpenSSL {SHA512_Init, SHA512_Update, SHA512_Final, Openssl_cleanse}, the sequence mined by the ml4spec method contains only the first three functions. In terms of ER, the ml4spec method is superior to the first stage of the proposed approach without removing interference APIs and is closer to the case of removing interference APIs. This is because the clustering method based on textual similarity can eliminate some of the interference APIs, but still introduces some frequent common APIs with similar function names with functional APIs, such as TIFFErrorExt, mbedtls_debug_print_msg, etc.

API relationship mining experiment results

The goal of our approach is to mine a specification that contains the semantic relationships among multiple APIs. The current approaches can only efficiently mine the semantic relationship between API pairs. APISan is a typical approach proposed by Yun et al. (Yun et al. 2016). In this section, we compare the performance of the proposed approach with that of the APISan approach at extracting relationships of API pairs.

Among the three semantic relationships of API pairs mined in this paper, APISan only mines control dependencies and does not analyze parameter sharing and data dependency relationships. The 45 debian packages using Openssl as shown in Table 4 were employed as test data, because of the absence of public benchmark in API relationship mining. The benchmark data is set to the set of APIs relationships extracted by the two approaches. If the minimum support degree of the API relationship set in the APISan approach is greater than or equal to 5, many valid relationships will be missed, whereas when it is equal to 1, many invalid accidental relationships will be reported. Therefore, in the experiment, the APISan approach's API relationship minimum support was set to 2, 3 and 4 and then compared with the RR and ER of the extraction results of our approach. The results are shown in Table 7. This table shows that the efficiency of our approach is significantly higher than that of the APISan approach, mainly because the results extracted by the APISan approach contain many invalid relationships related to frequent common APIs. To analyze impact of frequent common APIs to the relationship mining, we made comparison of the result of under-constrained symbolic execution with and without removing them in the sequence mined from the first stage. As show in Table 7, with the removing of frequent common APIs, the RR is not changed but the ER is significantly improved. The memory usage and the processing time is reduced accordingly. It shows that the removing of frequent common APIs is effective in reducing the redundant API relationships in the result.

Moreover, the RR of the proposed approach is significantly higher than the APISan approach. This is because the APISan only mines control dependencies and does not analyze parameter sharing and data dependency relationships. In addition, we found that of.

the total 38 rules, 11 rules were uniquely mined by the proposed approach, as compared with the APISan approach.

In terms of memory usage and time overhead, the proposed approach is clearly superior to the APISan approach. This is mainly because the proposed approach

Table 7 Experimental results of this approach and APISan approach

Indicators	APISan			Without FCA removal	With FCA removal
	th = 2	*th* = 3	*th* = 4		
RR(%)	71	61	58	95	95
ER(%)	68	78	85	82	91
Maximum memory usage (MB)	2544	2437	2305	424	251
Time (s)	542	445	356	322	296

45 debian packages using Openssl were employed as test data.th: threshold, RR: recall rate, and ER: effective rate

Table 8 specification mining results

Program	API sequences	Mined specifications	Correct specifications
libTIFF	1,534	43	21
OpenSSH	3,516	98	54
mbed TLS	705	68	29
OpenSSL	13,406	113	72
Putty	1,905	89	53
zlib	310	27	15

API sequences: the number of API sequences extracted in the first stage, where each path contains at least two or more APIs, support: the minimum support for frequent items selected for different open-source project

only performs relationship extraction analysis on the APIs in the frequent sequences mined from the first stage, which filters out frequent common APIs and other infrequent APIs. The APISan approach extracts and analyzes the relationships among all APIs. The searching space for the possible API-pair relationship is substantially increased, so the time and memory overhead are large. The APISan approach can only analyze partial code because of the large memory overhead. The lower memory and time overhead of our approach enables it to analyze the entire code of the target project.

In addition, we found that the number of rules extracted by the proposed approach is much lower than the number of APISan. This is because the rules extracted by the proposed approach contains the aggregation of the relationships of multiple APIs, while the rules extracted by APISan contains the pre/post conditions and return value dependencies between each API pair. There is no aggregation of multiple API semantic relationships, which would greatly increase the number of rules and the cost of subsequent vulnerability analysis. The final experiment combines the pairs of relationships and explores the usage specifications of multiple APIs. Table 8 shows the specification mining results for six open-source projects. The number of sequences extracted, the mining runtime, and the size of the code increase proportionally. The number of specifications is also consistent with the size of the code except that the number of specifications of mbed TLS-2.16.0 is less than that of OpenSSH-7.9 and that of mbed TLS-1.14.0 is less than that of OpenSSL-1.1.1. This is because the protocols implemented by OpenSSH-7.9 and OpenSSL-1.1.1 are more complex than those of the others.

Analysis of API call specifications violations

Counter examples generate after the frequent API sets and the API call specifications mining. During our experiment on OpenSSL, Putty, Gnutls and mbed TLS, we made further analysis on counter examples and found that violations of API call specifications, such as missing calls, missing checks, ignoring return values, cause security threats likely. API call models constructed by specifications contributes to automatic tools (CGF) to mine vulnerabilities. This paper selects two typical security threats caused by violations of API call specifications, including information leakage and access check bypass, and selects vulnerability examples to analyze and illustrate our thoughts.

Information leakage

Figure 13 shows the API call specifications of hash operations in Putty-0.70 and OpenSSL-1.1.1. When the hash function was called, the clean function needs to be called to perform the memory deallocation of the hash variable.

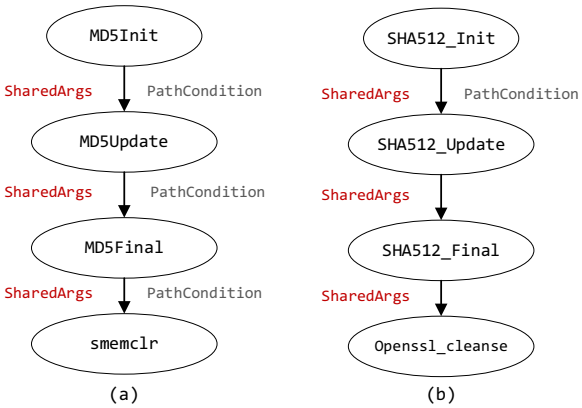


Fig. 13 API call specifications of hash operations in Putty and OpenSSL. **a** API call specification of hash operation in Putty. **b** API call specification of hash operation in OpenSSL

We reported our findings to the OpenSSL development team, and it was officially acknowledged. The issue has been fixed in later versions.

Access control bypass

Unckeck GnuTLS certificate

Gnutls is an OpenSSL-like implementation of the SSL protocol used in several projects such as pidgin, scrollz, and mod_gnutls. The SSL operating specifications is shown in Fig. 14. The specification is about how an SSL connection determines whether the opponent certificate is valid. If not invoked correctly, the certificate validation function will fail and a risk of "man-in-the-middle attack" exists by forged certificate.

Only in the current proxy mode in mod_gnutls module after calling gnutls_init, the gnutls_certificate_set_verify_ function function is called for access verification, while nothing is called for certificate validation in any other path. Such API calling sequence leads to attacks in the form of forged certificates.

Unchecked return value in mbed TLS

The program for the embedded system references the mbed TLS (formerly known as PolarSSL) library, an implementation of the embedded SSL protocol. The SSL operation specifications is shown in Fig. 15.

We found two violations of this API call specifications. The one is in dtls server module. The missing call of mbedtls_ssl_get_verify_result to verify the certificate of client after calling mbedtls_ssl_handshake will lead to potential SSL man-in-the-middle attacks. The other is in cert_app module, where the missing check of return value after calling mbedtls_ssl_close_notify will cause unilateral close of connection without consultation and affect the usability of the program.

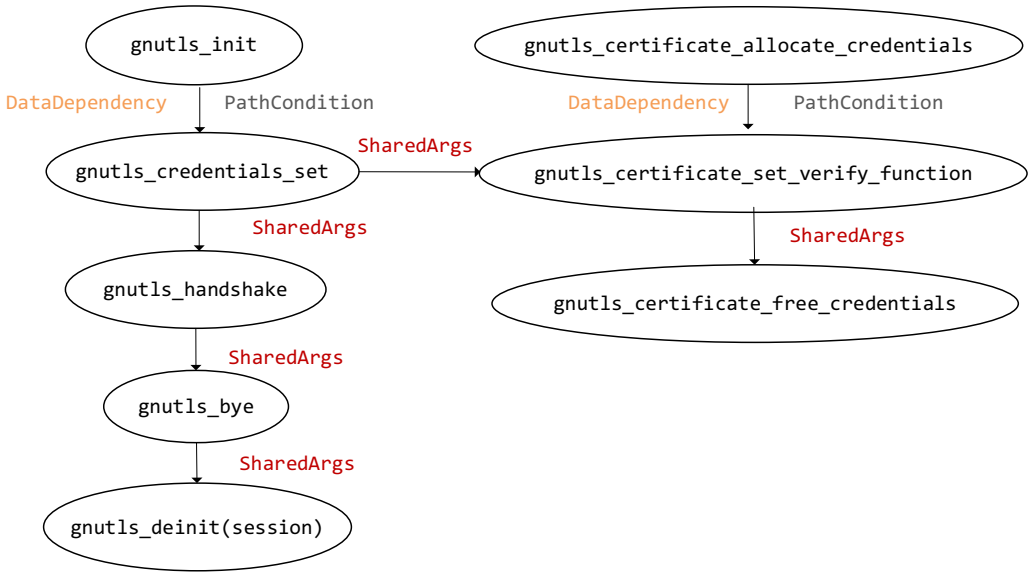


Fig. 14 API call specifications of SSL related in Gnutls

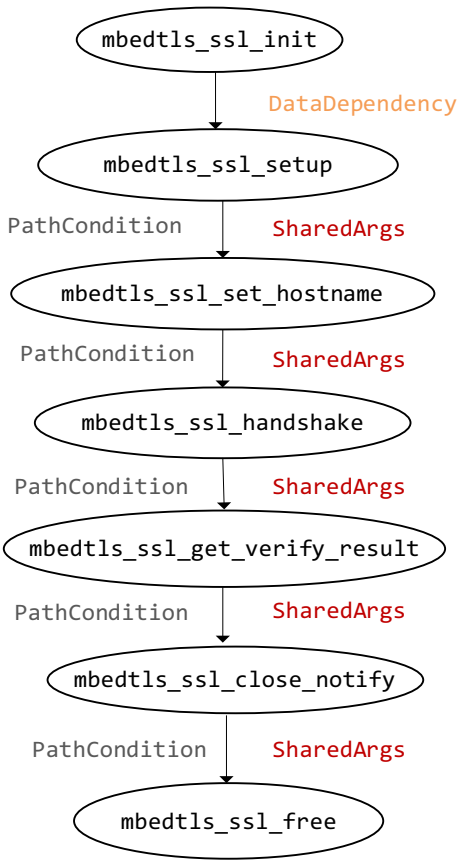


Fig. 15 API call specifications of SSL related in mbed TLS

The code of BN_generate_dsa_nonce function in OpenSSL-1.1.1 calls hash related API (Fig. 4). The code of pageant_handle_msg function in Putty calls hash related API (Fig. 16). Code of unchecked return value of mbedtls_ssl_close_notify function in cert_app module in mbed TLS (Fig. 17).

Conclusion

This paper proposed an API specification mining approach that efficiently extracts a relatively complete list of the API combinations and semantic relationships between APIs. The approach mines the target code in two stages. The first stage uses the improved maximum frequent item-set mining algorithm after frequent common API identification and filtration to obtain accurate frequent API sequences. Using the results of the first stage, the second stage employs a semantic relationship sensitive API specification automatic mining method based on domain adapted under-constrained symbolic execution and graph-based relationship aggregation to mine flow-, path-, and context-sensitive multiple API call specifications. The experimental results show that the proposed frequent itemset mining algorithm is superior to the classical PR-Miner approach in terms of efficiency and recall rate. For the final API call specification, not only is the performance of the proposed API-pair relationship mining better than that of the existing typical approach of APISan, but it can mine

```
putty-0.70\pageant.c
286 void *pageant_handle_msg(const void *msg, int msglen, int *outlen,
                           void *logctx, pageant_logfn_t logfn)
{
……
306     switch (type) {
……
376     case SSH1_AGENTC_RSA_CHALLENGE:
……
454      MD5Init(&md5c);
455      MD5Update(&md5c, response_source, 48);
456      MD5Final(response_md5, &md5c);
457      smemclr(response_source, 48);  /* burn the evidence */
458      freebn(response);           /* and that evidence */
459      freebn(challenge);          /* and that evidence */
474  break;…
}
……
892  return ret;
}
```

Fig. 16 The code of pageant handle msg function in Putty calls hash related API

```
466        mbedtls_printf( "%s\n", buf );
……
468        mbedtls_ssl_close_notify( &ssl );
470  ssl_exit:
471        mbedtls_ssl_free( &ssl );
472        mbedtls_ssl_config_free( &conf );
```

Fig. 17 Code of unchecked return value of mbedtls_ssl_close_notify function in cert app module in mbed TLS

multiple API call specifications. Moreover, the mining efficiency was also shown to be significantly improved.

Acknowledgements
We thank the anonymous reviewers for their helpful remarks. We thank the editor and the reviewers for their useful feedback that improved this paper.

Author contributions
Zhongxu Yin: Conceptualization of this study, Methodology, Validation. Yiran Song: Formal analysis, Data Curation. Guoxiao Zong: Investigation, Data Curation, Writing—Original draft preparation, Visualization.

Funding
No funding.

Availability of data and materials
All data generated or analyzed during this study are included in this published article.

Declarations

Competing interests
All authors disclosed no relevant relationships.

Received: 3 July 2023 Accepted: 20 February 2024
Published online: 03 October 2024

References

Bian P et al (2018a) Detecting bugs by discovering expectations and their violations. IEEE Trans Softw Eng 45(10):984–1001
Bian P et al. (2018) "Nar-miner: Discovering negative association rules from code for bug detection". In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. pp. 411–422.
Chang R-y, Podgurski A (2012) Discovering programming rules and violations by mining interprocedural dependences. J Softw: Evolut Process 24(1):51–66
Chang R-Y, Podgurski A, Yang J (2008) Discovering neglected conditions in software by mining dependence graphs. IEEE Trans Softw Eng 34(5):579–596
Chen L et al (2018) Automatic mining of security-sensitive functions from source code. Comput, Mater Continua. https://doi.org/10.3970/cmc.2018.02574
Dyer R et al. (2013) "Boa: A language and infrastructure for analyzing ultra-large- scale software repositories". In: 2013 35th international conference on software engineering (ICSE). IEEE. pp. 422–431.

Grahne G and Zhu J (2003) "Efficiently using prefix-trees in mining frequent itemsets." In: FIMI. Vol. 90 pp 65.
Grahne G and Zhu J (2003) "High performance mining of maximal frequent itemsets". In: 6th International workshop on high performance data mining. Vol. 16. pp 34.
He B et al. "Vetting SSL Usage in Applications with SSLINT". In: 2015 IEEE Symposium on Security and Privacy. 2015, pp. 519–534. doi: https://doi.org/10.1109/SP.2015.38.
Henkel J et al. (2019) "Enabling Open-World Specification Mining via Unsupervised Learning". In: arXiv preprint arXiv:1904.12098
Huan J et al. (2004) "Spin: mining maximal frequent subgraphs from graph databases". In: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. pp 581–586.
Jana S, Kang Y J, Roth S, et al. (2016) Automatically detecting error handling bugs using error specifications[C]//25th USENIX Security Symposium (USENIX Security 16). pp 345–362.
Kang Y, Ray B and Jana S . (2016) "Apex: Automated inference of error specifications for c apis". In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp 472– 482.
Karp RM and Tarjan RE . (1980) "Linear expected-time algorithms for connectivity problems". In: Proceedings of the twelfth annual ACM symposium on Theory of computing. pp 368–377.
Lee G et al. "Approximate maximal frequent pattern mining with weight conditions and error tolerance". In: International Journal of Pattern Recognition and Artificial Intelligence 30.06 (2016), p. 1650012.
Lee G, Yun U (2018) Performance and characteristic analysis of maximal frequent pattern mining methods using additional factors. Soft Comput 22:4267–4273
Lemieux C , Park D , and Beschastnikh I . (2015) "General LTL speci- fication mining (T)". In: 2015 30th IEEE/ACM international conference on automated software engineering (ASE). IEEE., pp 81–92.
Liang B et al. (2016) "AntMiner: mining more bugs by reducing noise interference". In: Proceedings of the 38th international conference on software engineering. pp 333–344.
Li Z, Zhou Y (2005) PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. ACM SIGSOFT Softw Eng Notes 30(5):306–315
Lv T, Li R, Yang Y, et al. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection[C]// Proceedings of the 2020 ACM SIGSAC conference on computer and communications security. 2020 pp 1837-1852
MicrochipTech. MicrochipTech mbedtls examples. https://github.com/MicrochipTech/mbedtls-examples. 2019.
Nguyen HA et al. (2014) "Mining preconditions of APIs in large-scale code corpus". In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. pp. 166–177.
Nguyen HA et al. (2015) "Consensus-based mining of API preconditions in big code". In: Companion Proceedings of the 2015 ACM SIGPLAN international conference on systems, programming, languages and applications: software for humanity. pp 5–6.
Ramanathan MK, Grama A , and Jagannathan S. (2007) "Static specification inference using predicate mining". In: ACM SIGPLAN Notices 42.6, pp 123–134.
Ramos DA and Engler D (2015) "Under-constrained symbolic execution: Correctness checking for real code". In: 24th USENIX Security Symposium (USENIX Security 15), pp 49–64.
Schlichtig M, Sassalla S, Narasimhan K, et al. (2022) Fum-a framework for api usage constraint and misuse classification[C]//2022 IEEE international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 673–684.
Shastry B et al. (2016) "Towards vulnerability discovery using staged program analysis". In: detection of intrusions and malware, and vulnerability assessment: 13th international conference, DIMVA 2016, San Sebasti´an, Spain, July 7–8, Proceedings 13. Springer. 2016, pp 78–97.
Tamaskar SD, Raut AB. Approach for Mining in Lossless Representation of Closed Itemsets[J]. 2016(11).
Wang X, Zhao L. APICAD: Augmenting API Misuse Detection through Specifications from Code and Documents[C]//2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023: 245–256.
Yamaguchi F, Wressnegger C, Gascon H, et al. Chucky: Exposing missing checks in source code for vulnerability discovery[C]//Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013: pp 499-510
Yin Z et al (2020) A security sensitive function mining approach based on precondition pattern analysis. Comput, Mater Continua 63(2):1013–1029
Yun I et al. (2016) "APISan: Sanitizing API Usages through Semantic Cross-Checking." In: Usenix Security Symposium. pp. 363–378.
Yun U, Lee G (2016) Incremental mining of weighted maximal frequent itemsets from dynamic databases. Expert Syst Appl 54:304–327
Yun U, Lee G, Lee K-M (2016) Efficient representative pattern mining based on weight and maximality conditions. Expert Syst 33(5):439–462

Publisher's Note