APP-Miner: Detecting API Misuses via Automatically Mining API Path Patterns

Jiasheng Jiang, Jingzheng Wu, Xiang Ling, Tianyue Luo, Sheng Qu, and Yanjun Wu*
Institute of Software, Chinese Academy of Sciences, China
{*jiasheng, jingzheng08, lingxiang, tianyue, qusheng, yanjun*}*@iscas.ac.cn*

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00043

***Abstract*—Extracting API patterns from the source code has been extensively employed to detect API misuses. However, recent studies manually provide pattern templates as prerequisites, requiring prior software knowledge and limiting their extraction scope. This paper presents APP-Miner (API path pattern miner), a novel static analysis framework for extracting API path patterns via a frequent subgraph mining technique without pattern templates. The critical insight is that API patterns usually consist of APIs' data-related operations and are commonplace. Therefore, we define API paths as the control flow graphs composed of APIs' data-related operations, and thereby the maximum frequent subgraphs of the API paths are the probable API path patterns. We implemented APP-Miner and extensively evaluated it on four widely used open-source software: Linux kernel, OpenSSL, FFmpeg, and Apache httpd. We found 116, 35, 3, and 3 new API misuses from the above systems, respectively. Moreover, we gained 19 CVEs.**

***Index Terms*—API misuse, specification inference, control flow graph, frequent subgraph mining, static analysis**

1. Introduction

Nowadays, it is common practice for programmers to use APIs to realize specific functions without understanding the details of the internal working mechanism, which indicates the API patterns. However, violating API patterns will cause API misuses and can have profound security implications [18], [22], [32]–[34], [36], [45]–[47], [50], [54]. A well-known example is the pattern of *kmalloc*-class APIs (e.g., kmalloc, kvmalloc, and kcalloc): {*check allocation size* → *kmalloc* → *check return pointer* → *free*}. A Remote Stack Overflow vulnerability is expected if a function implementation misses the *check allocation size* [5], [6]. Besides, if *free* is missed, it will cause a Memory Leak vulnerability [7]. Moreover, we scrutinized 92 Linux kernel vulnerabilities (published in 2021) in CVE details [15] and found that 27 (29.3%) stem from API misuses.

In detecting API misuses, the critical challenge is to gain the corresponding API patterns. Existing researchers for extracting API patterns can be broadly classified into three categories. The first category focuses on extracting API patterns from the source code [19], [24], [27], [35], [37], [39], [44], [49], [52], [53]. Most of them manually provide pattern templates as prerequisites and try to discover in what contexts the APIs should be used as templates. However, their templates require prior knowledge of the software and limit their extraction scope. In addition, FuzzGen [27] presumes that all the usages of APIs in real programs are valid and coalesces them as the API patterns. However, they only sometimes hold. The second category generates the test cases to execute the software and monitor the dynamic tracing to mine the frequent sequences of APIs as the API patterns [20], [23], [28], [30], [31], [38], [40]–[43]. Nevertheless, generating test cases needs much manual effort. Moreover, because of the low coverage rate, it is challenging for the work to gain complete API patterns. The third category extracts the patterns from the documentation using NLP technologies [36], [48], [54]. Unfortunately, most API patterns are too tedious to be documented by programmers, which are implicit and hidden in the source code [32].

This paper presents `APP-Miner` (API path pattern miner), a novel static analysis framework for extracting API path patterns via a frequent subgraph mining technique from the source code without pattern templates. Unlike the first category, our goal is to discover the probable correct usages of the APIs. The critical insight is that API patterns usually consist of APIs' data-related operations and are commonplace in the source code. Therefore, if we build control flow graphs consisting of APIs' data-related operations as API paths, the maximum frequent subgraphs of the API paths are probable API path patterns. After extracting the API path patterns, we detect the violations as potential API

*Corresponding author

DOI 10.1109/SP54263.2024.00043

misuses, which do not contain the API path patterns as subgraphs. Note that some APIs have more than one path pattern. In these cases, using the APIs should obey one of the path patterns. It is reasonable as some APIs have different correct usages, using any of which is correct.

However, extracting API path patterns has several challenges. (1) Infrequent nodes between the frequent nodes result in unconnected frequent subgraphs, while the API path patterns should be connected. (2) The frequent subgraph mining is an exponential time complexity process that does not apply to large software. To address the first challenge, `APP-Miner` adds edges from each node to its descendants. The newly added edges connect the frequent nodes to make the frequent subgraphs connected. To avoid path explosion, `APP-Miner` unrolls the loops only once before at first. In a word, the API paths are converted from general digraphs into complete topologies. To address the second challenge, `APP-Miner` first aligns the nodes and builds index matrixes that contain forward and inverted indexes to store the mappings between edges and API paths. As a result, it allows `APP-Miner` to use binary arithmetic to accomplish graph operations, thereby significantly speeding up the frequent subgraph mining. Second, `APP-Miner` utilizes the *downward closure property* [17], [25], [26], [29], [51] that states subgraphs of frequent graphs must be frequent. It is known that if the statement is true, then its contrapositive is true [14]. Therefore, if the subgraph is not frequent, any graph containing this subgraph must not be frequent. Specifically, `APP-Miner` generates each k-edge frequent subgraph by combining two (k-1)-edge frequent subgraphs with (k-2) same edges. Since most subgraphs are infrequent, `APP-Miner` eliminates exponentially redundant candidates and can effectively extract API path patterns from large software. In summary, we propose a new method that contains topologization, completion, index matrix building, and redundancy removal to help extract API path patterns correctly and efficiently.

We implemented `APP-Miner` and chose four widely used open-source software to evaluate our method extensively: Linux kernel, OpenSSL, FFmpeg, and Apache httpd. `APP-Miner` extracted 4,788 API path patterns from the first three. Then, we used these API path patterns to detect their respective violations. Besides, we use the API path patterns from the OpenSSL to detect the API misuses in Apache httpd since it has an "ssl" module that uses the OpenSSL library APIs. After that, we manually checked the top 500, 100, 20, and 20 violations and submitted patches. Up to July 12th, 2023, we found 116, 35, 3, and 3 new API misuses from the above systems, respectively. Moreover, we gained 19 CVEs. The results confirm the effectiveness of `APP-Miner` in extracting API patterns and detecting API misuses. In summary, the key contributions of this paper are:

- **A novel API path pattern extraction framework.** We propose a novel static analysis framework to automatically extract API path patterns via a frequent subgraph mining technique without manual efforts to provide pattern templates.
- **A new method to deal with the challenges of extracting API path patterns.** `APP-Miner` converts the API paths into complete topologies that help extract correct API path patterns. Besides, `APP-Miner` builds index matrixes and removes redundant candidates to improve extraction efficiency.
- **Finding and fixing numerous new bugs.** Utilizing `APP-Miner`, we found multiple bugs in Linux kernel, OpenSSL, FFmpeg, and Apache httpd, which could cause various security vulnerabilities. We submitted patches to fix these bugs, and 157 were successfully applied to the master branches. Moreover, we gained 19 CVEs.

The source code of our prototype is available at https://github.com/JiangJias/APP-Miner, allowing other researchers to extend our automatic framework to extract API path patterns.

The rest of the paper is organized as follows. §2 provides the motivation of `APP-Miner`. §3 and §4 present system design and implementation details. We evaluate the effectiveness of `APP-Miner` in §5 and discuss the limitations of `APP-Miner` in §6. Finally, we present related work in §7 and conclude the paper in §8.

2. Motivation

In this section, we motivate our framework by a misuse of *_usecs_to_jiffies* in Linux kernel (v5.15-rc7). The pattern and misuse of *_usecs_to_jiffies* are shown in Figure 1(a) and Figure 1(c). The *_usecs_to_jiffies* can convert *microseconds* into *jiffies*. The calculated equation is as follows:

$$jiffies = (M * microseconds + A) >> S \quad (1)$$

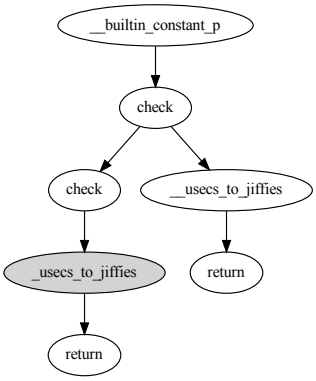
The *M*, *A*, and *S* are constants, and *jiffies* and *microseconds* are unsigned integers. It can be found that when *microseconds* is not limited, an Integer Overflow error may occur.

It is a challenge for existing studies to get the pattern.

- **Source code:** Crix [35] collects 531 error-handling functions to identify the security

```
1  /* include/linux/jiffies.h */
2  static __always_inline unsigned long
   ↪  usecs_to_jiffies(const unsigned
   ↪  int u)
3  {
4    if (__builtin_constant_p(u)) {
5      if (u > jiffies_to_usecs(
       ↪  MAX_JIFFY_OFFSET))
6        return MAX_JIFFY_OFFSET;
7      return _usecs_to_jiffies(u);
8    } else {
9      return __usecs_to_jiffies(u);
10   }
11 }
```

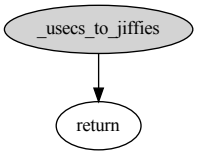
(a) A pattern of *_usecs_to_jiffies* in *usecs_to_jiffies*



(b) The path pattern of *_usecs_to_jiffies*

```
1  /* net/rxrpc/rtt.c */
2  static u32 __rxrpc_set_rto(const
   ↪  struct rxrpc_peer *peer)
3  {
4    return _usecs_to_jiffies((
     ↪  peer->srtt_us >> 3) +
     ↪  peer->rttvar_us);
5  }
```

(c) A misuse of *_usecs_to_jiffies* in *__rxrpc_set_rto*



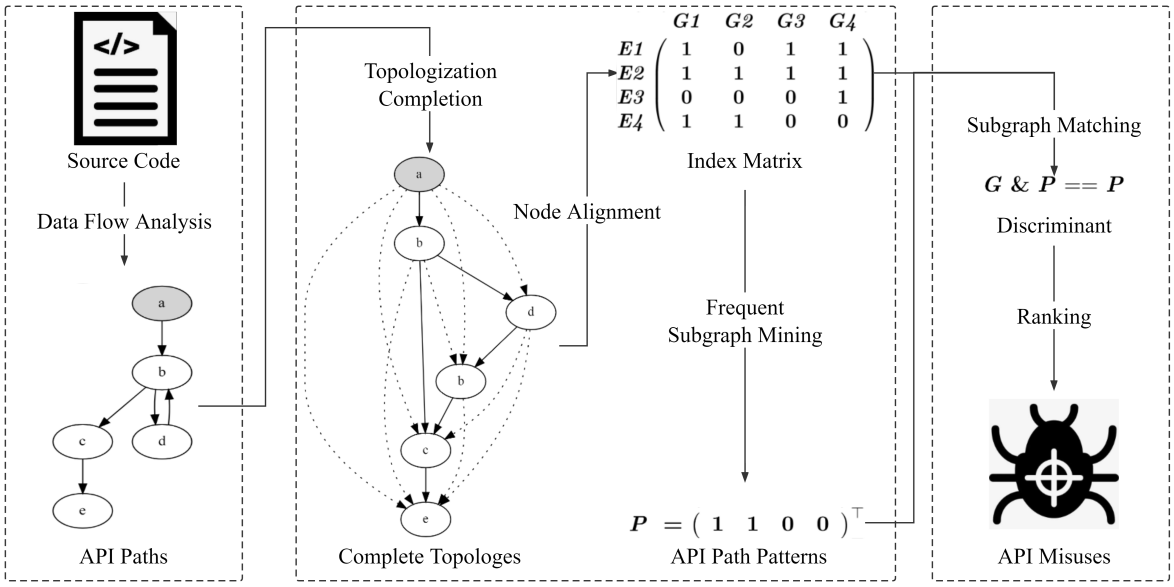
(d) The violated path of *_usecs_to_jiffies*

Figure 1: An example of API misuse in Linux kernel, detected by APP-Miner. APP-Miner extracts the API path pattern since 146 out of 147 usages of *_usecs_to_jiffies* follow the pattern shown in (a) and (b). Applying the API path pattern, APP-Miner detects an API misuse shown in (c) and (d).

checks that APIs may miss. However, when the check for *_usecs_to_jiffies* fails, it will return a max macro rather than the error-handling functions. APISan [52] extracts the frequent API usages in the same symbolic contexts as the API patterns. However, most of the usages of the *_usecs_to_jiffies* are in different contexts. PR-Miner [32] uses a frequent itemset mining technique to extract the functions and variables frequently appearing together. Not only does it fail to add condition checks as the items, but the itemset is unordered. As a result, it cannot distinguish whether the condition check should be before or after the API. For example, it may extract the pattern {*check*, *_usecs_to_jiffies*}, the same as the misuse: { *_usecs_to_jiffies*, *check*}.

- **Dynamic tracing:** Pradel and Gross [40], [41] produce FSMs describing legal sequences of method calls during the executions. Nevertheless, they only focus on the Iterator from the Java standard library. Furthermore, there are two checks in the API pattern and 29 checks on the path from the system call interface to the misuse code. These path branches hinder the extraction of the complete pattern and the detection of misuse, respectively.
- **Documentation:** Advance [36] detects API misuses by verifying derivation from library documentation. However, the *_usecs_to_jiffies* is undocumented in the Linux kernel API documentation [16].

We have observed a critical insight that 146 out of 147 usages of *_usecs_to_jiffies* follow the pattern in Figure 1(a). Moreover, we have found that the critical operations are data-related to the APIs. For example, the second check checks the parameter of *_usecs_to_jiffies* to avoid Integer Overflow in Figure 1(a). Therefore, we can build control flow graphs consisting of operations data-related to *_usecs_to_jiffies* from the 147 usages, and thereby, the maximum frequent subgraphs are the patterns.



(a) API Path Generation (b) API Path Pattern Extraction (c) API Misuse Detection

Figure 2: APP-Miner overview. The black node means the API, and the white nodes are the operations that are data-related to the API.

Thus, we present APP-Miner, which uses the frequent subgraph mining technique to automatically extract the patterns from the source code. It considers all the APIs' data-related operations, whether bound checks, memory allocations, or read/write locks. Therefore, it can find more complex and unknown API path patterns uncovered by the existing studies.

3. APP-Miner

3.1. Overview

APP-Miner is a static analysis framework that aims to detect API misuses via automatically mining API path patterns. The high-level idea of APP-Miner is to employ a frequent subgraph mining technique to infer API path patterns to detect violations. Moreover, APP-Miner regards all function calls as APIs. Because similar to APIs, names of functions are usually meaningful and bound to patterns of functions.

However, extracting API path patterns has several challenges: (1) Infrequent nodes between frequent nodes make frequent subgraphs unconnected while API path patterns should be connected, and (2) the frequent subgraph mining is an exponential time complexity process that does not apply to large software. We propose a new method to extract API path patterns correctly and efficiently according to the challenges.

Figure 2 presents the overview of APP-Miner. It starts from the source code and uses the data flow analysis to generate the API paths (§3.2). Then, APP-Miner employs topologization and completion to deal with the first challenge. After that, it builds index matrixes and uses a frequent subgraph mining algorithm based on the *downward closure property* [17], [25], [26], [29], [51] to extract the API path patterns efficiently (§3.3). At last, APP-Miner uses a binary discriminant to match the subgraphs and ranks the violations to find the probable API misuses (§3.4).

3.2. API Path Generation

API paths are the control flow graphs consisting of APIs' data-related operations. API paths contain three types of program elements: APIs, condition checks, and return statements. The APIs and condition checks are the primary sources of API misuses. For example, we scrutinized 27 API misuses of Linux kernel vulnerabilities (published in 2021) in CVE details [15] and found that 25 resulted from missing or disordered APIs or condition checks. Furthermore, since most return values are checked after the calls, APP-Miner regards return statements as condition checks.

Like many studies in mining patterns from the source code [19], [21], [32], [52], APP-Miner generates the API path intraprocedurally. Specifically,

Algorithm 1: Convert into Complete Topology

```
Input: GDSet: General digraph set
Output: CTSet: Complete topology set
CTSet ← ∅
for GD ∈ GDSet do
    CT ← ∅
    for Path ∈ Topologization(GD) do
        CT ← CT + Completion(Path)
    end
    CTSet ← CTSet ∪ {CT}
end

Procedure Topologization(GD)
abstract ← ∅, stack ← ∅, node ← GD.root()
abstract[node] ← ∅, stack.push(node)
while stack not empty do
    node ← stack.pop()
    for child in node.child() do
        abs ← abstract[node].tail() ∪ {node}
        if Set(abs) in Set(abstract[child]) then
            continue
        end
        abstract[child].append(abs)
        stack.push(child)
    end
end
// The abstract[GD.tail()] records all the paths
   from the entry to the end of the GD
return abstract[GD.tail()]

Procedure Completion(Path)
CT ← ∅
for i in [0, Path.length() - 1] do
    for j in [i, Path.length() - 1] do
        CT.append(Path[i] → Path[j])
    end
end
return CT
```

APP-Miner first uses the Clang compiler to convert each function implementation into a control flow graph. The LLVM provides many interfaces to help data flow analysis. Then, APP-Miner uses LLVMGetNextBasicBlock [10] and LLVMGetNextInstruction [11] to traverse the control flow graph. APP-Miner generates paths for each API during the traversal by the following three steps:

The first step is recursively using LLVMGetOperand [12] to get sources that assign the values to the API parameters.

The second step is recursively using LLVMGetUser [13] to obtain the API's data-related operations that use the sources.

The third step uses the API's data-related operations to generate the API path. Specifically, APP-Miner connects operations like $\{a \rightarrow b\}$ if a pathway from a to b does not pass through the API's data-related operations in the control flow graph.

For example, the API path of *_usecs_to_jiffies* in Figure 1(b) is generated from the source code in Figure 1(a). In the first step, APP-Miner collects the source $\{u\}$, the function argument. In the second step, APP-Miner gains the API's data-related operations {*__builtin_constant_p*, *check*, *check*, *_usecs_to_jiffies*, *return*, *__usecs_to_jiffies*, *return*}, the user operations of $\{u\}$. In the third step, APP-Miner connects the operations by the execution order.

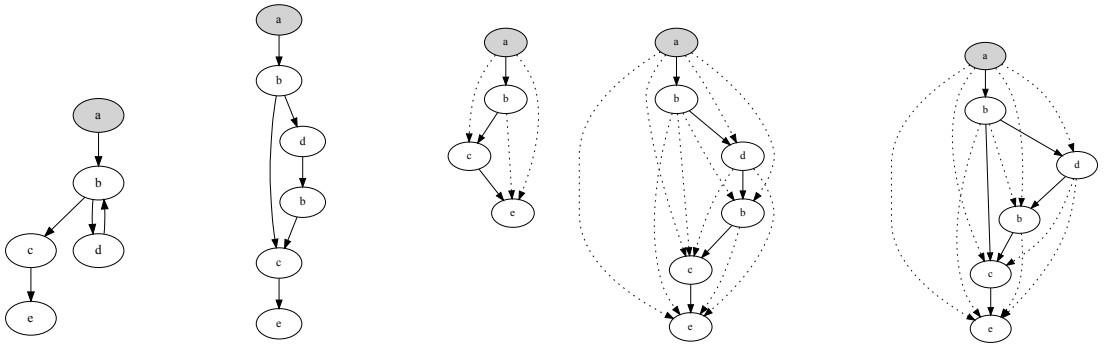
However, there still exists imprecise data flow analysis that can challenge generating complete API paths. For example, nested structures may exist, containing multiple pointers that cannot be dealt with by LLVM interfaces. Nonetheless, APP-Miner can still generate many API paths unaffected by the imprecise data flow analysis, thereby extracting the API path patterns by the frequent subgraph mining technique.

3.3. API Path Pattern Extraction

APP-Miner extracts API path patterns in a new method to address the challenges. First, it converts the API paths from general digraphs into complete topologies. Then, it aligns the nodes and builds index matrixes that contain forward and inverted indexes to store the mappings between edges and their locations in API paths. Last, it utilizes a frequent subgraph mining technique based on *downward closure property* to extract the maximum frequent subgraphs. The whole process is shown in Figure 2(b).

3.3.1. Topologization and Completion. In this part, APP-Miner addresses the challenge that infrequent nodes may exist between the frequent nodes in the API paths, which makes the extracted API path patterns unconnected. In Figure 3(a), assuming a, b, and e are frequent nodes while c is infrequent. Therefore, c and the edges related to c are removed during the frequent subgraph mining. As a result, the extracted API path pattern is $\{a \rightarrow b, e\}$, while the API path pattern should be connected.

To address the problem, APP-Miner uses topologization and completion techniques to convert API paths from general digraphs into complete topologies, unrolling the loops only once and adding assisted edges from each node to its descendants. Therefore, the newly added edges can connect the frequent nodes and help



(a) API path (b) Topologization (c) Completion (d) Complete topology

Figure 3: An example of converting an API path from a general digraph into a complete topology

extract the connected API path patterns. Moreover, according to the survey of 27 API misuses of Linux kernel (published in 2021) in CVE details [15], none is caused by calling the particular function only once. Therefore, it is acceptable for APP-Miner to eliminate the loops. The details are as follows:

- **Topologization:** APP-Miner unrolls the loops only once to convert API paths from general digraphs into topologies. Specifically, APP-Miner traverses each API path by depth-first strategy, recording the preorder nodes as abstracts for each node. Every time before visiting a node, APP-Miner will check whether the node has been visited with the same abstract before, stopping and deleting the visiting process if yes.
- **Completion:** After the topologization, APP-Miner adds assisted edges from each node to its descendants that turn API paths into complete graphs. As a result, the assisted edges connect frequent nodes, and the extracted frequent subgraphs are connected.

Algorithm 1 shows converting API paths from general digraphs into corresponding complete topologies. APP-Miner first performs Topologization on each *GD* in *GDSet* to convert it into topology without loops (line 4). Then, APP-Miner performs Completion on each *Path* of the topology, adding assisted edges from each node to its descendants. All processed paths are aggregated into *CT* (line 5). Finally, APP-Miner appends the *CT* to the *CTSet* (line 7).

Figure 3 shows an example of converting an API path from a general digraph into a complete topology. There is a loop "$b \leftrightarrows d$" in the API path. APP-Miner first traverses the API path in Figure 3(a) by depth-first strategy and records the abstract of each node. When APP-Miner first and second visits d, the abstract of d is $\{a, b\}$ and $\{a, b, d\}$, respectively. However, the abstract of the third visit is still $\{a, b, d\}$, which is the same as the second visit. APP-Miner stops and deletes the third visiting progress. Therefore, the loop is unrolled only once, and the API path is converted into topology, as shown in Figure 3(b). After that, APP-Miner adds assisted edges for the two paths in the topology, which are shown as dotted lines in Figure 3(c). Finally, APP-Miner aggregates the two paths into one complete topology, as shown in Figure 3(d).

During the frequent subgraph mining, APP-Miner can now add assisted edges $\{a \rightarrow e\}$ and $\{b \rightarrow e\}$ into the API path pattern. Consequently, the extracted API path pattern is $\{a \rightarrow b \rightarrow e\}$.

3.3.2. Building Index Matrix. In this part, APP-Miner builds the index matrixes to store the forward and inverted indexes to improve the efficiency of frequent subgraph mining and subgraph matching.

APP-Miner uses the API names, "check", and "return" to label the nodes. Since parameters can infect the APIs' behaviors, we use the number of parameters as the nodes' second label to distinguish the APIs with the same name. Then, APP-Miner can label the edges by their head and tail nodes. For example, if two edges have the same head and tail nodes, they will be regarded as the same.

After the alignment, each graph can be seen as a set of edges. Therefore, we can use the set arithmetic to accomplish the graph operations, shown as "Set Arithmetic" in Table 1. For example, "Location(S_i, S_j)" means the set of API paths containing both subgraphs

TABLE 1: The set and binary arithmetic to accomplish graph operation

Graph Operation	Set Arithmetic		Binary Arithmetic	
	Arithmetic	Time	Arithmetic	Time
Merge(S_i, S_j)	Union(S_i, S_j)	$\mathcal{O}(N^2)$	S_i **OR** S_j	$\mathcal{O}(1)$
Location(S_i, S_j)	Intersection(Location(S_i), Location(S_j))	$\mathcal{O}(N^2)$	InvertedIndex(S_i) **AND** InvertedIndex(S_j)	$\mathcal{O}(1)$
Support(S)	Crad(Location(S))	$\mathcal{O}(N)$	InvertedIndex(S).count()	$\mathcal{O}(N)$
IsSubgraph(G, S)	IsSubset(G, S)	$\mathcal{O}(N^2)$	G **AND** S == S	$\mathcal{O}(1)$

S_i and S_j. We can now use "intersection" to gain the results.

Moreover, `APP-Miner` builds an index matrix to accelerate the process. At first, `APP-Miner` builds the lexicons of API paths and edges. After that, `APP-Miner` traverses all the API paths and uses "0/1" to indicate whether the edge exists in the API path. As a result, each column represents a forward index, and each row represents an inverted index. For example, *G1* represents the API path in Figure 2(a), and *E1* represents $\{a \rightarrow b\}$. The first row is "1011", meaning *E1* is contained in *G1*, *G3*, and *G4*.

As a result, we can replace the set arithmetic with binary arithmetic, shown as "Binary Arithmetic" in Table 1.

First, when merging two subgraphs, we utilized "Union" to combine the edges from the subgraphs and remove the duplicate edges that check whether every edge in S_i is in S_j. It is a square complexity arithmetic. We can do it by performing the "**OR**", which is constant complexity.

Second, we needed to use "Intersection" to gain the common location of the subgraphs. Similar to the "Union", it is square complexity. Now, the "Location(*S*)" is the inverted index of *S*. We can use a constant complexity "**AND**" to accomplish the same goal.

Third, we can count how many "1"s are in the inverted indexes of subgraphs to get support, which will be discussed in §3.3.3.

Last, instead of checking whether all the edges of the *S* are in the *G*, we can utilize the discriminant "*G* **AND** *S* == *S*" to judge whether *S* is the subgraph of the *G*. Similarly, the latter's complexity is constant, faster than the former's square.

The arithmetic will be performed exponentially during frequent subgraph mining. Benefiting from the index matrix, `APP-Miner` can do three types of them within $\mathcal{O}(1)$ time complexity rather than $\mathcal{O}(N^2)$.

3.3.3. Frequent Subgraph Mining. To extract frequent subgraphs, a trivial method is to check every subgraph. However, each graph has an exponential number of subgraphs. For example, an n-node graph has $\sum_{i=1}^{n} C_n^i = 2^n$ subgraphs. Therefore, directly generating all the subgraphs and counting the frequency is unsuitable for large software. Since most subgraphs are infrequent, we should eliminate the candidate subgraphs as much as possible. As previous work [17], [25], [26], [29], [51] shows, the frequent subgraph obeys the *downward closure property* that states subgraphs of frequent graphs must be frequent. It is known that if the statement is true, then its contrapositive is true [14]. Therefore, we can utilize the contrapositive that if a subgraph is infrequent, any graph containing this subgraph must be infrequent. Therefore, the graphs containing the infrequent subgraphs will not be considered candidates for API path patterns. In other words, `APP-Miner` only generates k-edge candidates from (k-1)-edge frequent subgraphs. Since most subgraphs are infrequent, `APP-Miner` can eliminate substantial redundant candidates and effectively extract API path patterns from large software.

During the frequent subgraph mining, `APP-Miner` should set two critical parameters: *min_support* and *confidence_threshold*, which means the minimum number of occurrences and ratio. Specifically, we use Support(*S*) to represent the number of API paths that contain *S*. For each *candidate* of the API path pattern, it is frequent if Support(*candidate*) is higher than *min_support* and Support(*candidate*) / Support(*API*) is higher than *confidence_threshold*. These two conditions must be met simultaneously to avoid extreme situations. It may happen if the first one is unsatisfied: Both the *API* and the *candidate* appear only once. If vice versa, a 100-support *candidate* may become the path pattern of a 10,000-support *API*.

Algorithm 2 shows the process of extracting API path patterns. `APP-Miner` first performs `Frequent` on $CSet_1$ to gain the $FSSet_1$ (line 2). Afterward, `APP-Miner` performs `Candidate` on the $FSSet_{k-1}$ to gain the $CSet_k$ (line 5). Then, `APP-Miner` performs `Frequent` on $CSet_k$ to gain $FSSet_k$ (line 6). Last, `APP-Miner` appends the $FSSet_k$ to *APPSet* and removes the subgraphs of $FSSet_k$ in *APPSet* (line 7). The k keeps adding one, and this process (lines 5-8) is repeated until FS_{k-1} is empty to gain the maximum frequent subgraphs as API path patterns. For example, the "$P = (1, 1, 0, 0)^T$" in Figure 2(b) represents an

Algorithm 2: Extract API path pattern

Input: $CSet_1$: 1-edge candidate set
Output: *APPSet*: API path pattern set

APPSet ← ∅
$FSSet_1$ ← Frequent($CSet_1$)
k ← 2
while $FSSet_{k-1}$ **do**
  $CSet_k$ ← Candidate($FSSet_{k-1}$)
  $FSSet_k$ ← Frequent($CSet_k$)
  APPSet ← *APPSet* ∪ $FSSet_k$ - Subgraph($FSSet_k$)
  k ← k + 1
end

Procedure Frequent($CSet_k$)
$FSSet_k$ ← ∅
for C_k *in* $CSet_k$ **do**
  if InvertedIndex(C_k).count() ≥ *min_support* **and** InvertedIndex(C_k).count() / InvertedIndex(*API*).count() ≥ *confidence_threshold* **then**
    $FSSet_k$.append(C_k)
  end
end
return $FSSet_k$

Procedure Candidate($FSSet_{k-1}$)
$CSet_k$ ← ∅
for $\{FS^m_{k-1}, FS^n_{k-1}\}$ *in* $FSSet_{k-1}$ **do**
  C_k ← FS^m_{k-1} **OR** FS^n_{k-1}
  InvertedIndex(C_k) ← InvertedIndex(FS^m_{k-1}) **AND** InvertedIndex(FS^n_{k-1})
  if $C_k \notin CSet_k$ **then**
    $CSet_k$.append(C_k)
  end
end
return $CSet_k$

extracted API path pattern {*E1*, *E2*}.

3.4. API Misuse Detection

Benefiting from the index matrix, APP-Miner can use the binary arithmetic discriminant "$G\&P == P$" to realize the subgraph matching to check the violations efficiently, as shown in Figure 2(c). Specifically, G means the API path, and P means the API path pattern.

Moreover, some APIs have more than one path pattern. In these cases, it is a violation only if it does not contain any of the path patterns. It is reasonable that the APIs have more than one correct usage.

After detecting violations, APP-Miner ranks them to find probable API misuses. As previous work [32], [35], [52] shows, patterns with higher occurrences will be more believable. In other words, the supports of violated API path patterns can be considered the possibility that the violations are API misuses. Therefore, the violations of API path patterns with higher support are more likely to be true API misuses than the lower ones. In addition, if the API has more than one path pattern, APP-Miner ranks the violations by the one with the highest support. After the ranking, we manually check the violations at the top and submit patches to the maintainer communities.

4. Implementation

4.1. Overview

We implement APP-Miner based on the LLVM. Specifically, we first use the Clang compiler to convert the source code into LLVM IRs. After that, we use LLVM interfaces to generate API paths. The details are as follows.

4.2. Intermediate Representation

As large software is too tedious to compile manually, we use the deadline [9] to compile C files into LLVM IRs automatically. Specifically, the tool compiles the target software by the Makefiles and records the commands. After that, the tool replaces the "gcc" or "g++" with "clang -S -emit-llvm" in each command. Moreover, it removes some unsupported options to compile successfully. At last, the tool runs the modified commands and generates the LLVM IRs, having convenient interfaces to help data flow analysis.

4.3. Data Flow Analysis

To deal with the LLVM IRs, we write an LLVM pass to traverse the control flow graphs. As mentioned earlier, APP-Miner focuses on three types of instructions: APIs, condition checks, and return statements. Specifically, we regard CallInst in the IRs as APIs, including direct and indirect calls. The second type includes BranchInst, SwitchInst, and SelectInst. The third type is ReturnInst. Moreover, BranchInst and SwitchInst should have multiple successors, which means they are not unconditional jumps (e.g., goto statement).

APP-Miner traverses the control flow graphs from roots to ends, adding operations to API paths if they use APIs' data-related variables and belong to the above three types. Notice that LoadInst and StoreInst represent

TABLE 2: Software evaluated in our experiments

Software	Version	# C files	# APIs	# API usages
Linux kernel	5.15-rc7	20,807	138,414	2,165,524
OpenSSL	3.0.0	1,301	6,962	111,019
FFmpeg	5.0	1,975	5,550	117,636
Apache httpd	2.4.52	176	2,091	22,498

reading and writing memory, which may cause indirect data flow. To address the problem, APP-Miner records the memory pointers that StoreInst uses to store APIs' data-related variables. If LoadInst reads the recorded pointers, APP-Miner will treat the return values as APIs' data-related variables.

5. Evaluation

5.1. Overview

In order to prove effectiveness, APP-Miner is evaluated to answer the following questions:

- Q1: How effectively does APP-Miner extract API patterns and find API misuses? (§5.2)
- Q2: How many API patterns and misuses found by APP-Miner are uncovered by state-of-the-art tools? (§5.3)
- Q3: What causes the false positives and false negatives of APP-Miner? (§5.4)
- Q4: What is the performance overhead of APP-Miner when analyzing the software? (§5.5)

Experiment Setup: We have evaluated APP-Miner with Linux kernel 5.15-rc7, OpenSSL 3.0.0, FFmpeg 5.0, and Apache httpd 2.4.52, shown in Table 2. Note that we filtered out the files that need more than one hour of data flow analysis. The filtered files only account for a small part. For example, there are 68 excluded files in Linux kernel, which comprise about 0.3% of the total. The experiments were performed on a virtual machine with 48 cores and 128GB RAM. Moreover, we use LLVM-10.0.0 to compile the software.

5.2. Extracting API Path Patterns and Finding API Misuses

We empirically study the former work using frequent mining in the source code [21], [32], [33], [47], [52]. As a result, we set *min_support* to 10 and *confidence_threshold* to 90%, similar to the former work.

APP-Miner first extracted API path patterns from Linux kernel, OpenSSL, and FFmpeg. After that, it utilized their respective patterns to detect violations.

TABLE 3: The experiment results of APP-Miner, including the extracted API path patterns (APPs), the API path patterns having violations (Violated APPs), and the detected violations (Violations)

Software	# APPs	# Violated APPs	# Violations
Linux kernel	3,985	2,009	18,593
OpenSSL	598	307	1,702
FFmpeg	205	125	1,189
Apache httpd	-	28	96
Total	4,788	2,469	21,580

Moreover, it utilized API path patterns from OpenSSL to detect the violations in Apache httpd since it uses a lot of OpenSSL APIs in the module called "ssl".

Table 3 shows the experiment results of APP-Miner in the evaluated software. We can find from the first column that APP-Miner can automatically extract 4,788 API path patterns from the first three of the evaluated software, requiring much work for programmers to specify and provide templates manually. Not only that, but the extracted API path patterns can help fuzz the library APIs [27].

The second column shows that about 1/2 of APPs have been violated, and the third column shows that APP-Miner has found many violations. We have manually examined the evaluated software's top 500, 100, 20, and 20 violations. Specifically, we check the violations to filter out the false positives, which the reasons in §5.4 will cause. Confirmed bugs have been reported to the corresponding developer community. Up to July 12th, 2023, 116, 35, 3, and 3 have already been fixed in the master branch. The detailed information is shown in Table 9-13 in the Appendix A. The results confirm the effectiveness of APP-Miner in finding API misuses. Moreover, the experiment result of Apache httpd proves the generality of the API path patterns extracted by APP-Miner.

Figure 4 shows the existing time of the confirmed API misuses detected by APP-Miner. For example, the bar with coordinates (1, 5) represents that five API

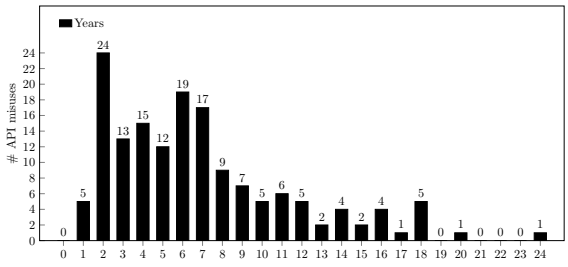


Figure 4: Existing time of accepted API misuses

TABLE 4: Types of accepted API misuses

Software	NULL Pointer Dereference	Uncaught Exception	Memory Leak	Missing Lock Check	Integer Overflow	Unused Variable	Total
Linux kernel	57	55	3	0	1	0	116
OpenSSL	29	2	1	2	0	1	35
FFmpeg	2	1	0	0	0	0	3
Apache httpd	1	2	0	0	0	0	3
Total	89	60	4	2	1	1	157

misuses exist between one and two years. We can find that the most extended period is more than 24 years, and 100 (63.69%) have existed for over five years.

We further investigated the types of accepted API misuses, shown in Table 4. We found that NULL Pointer Dereference accounts for 89 (56.69%) and Uncaught Exception accounts for 60 (38.22%), the main types of API misuses. Both of them can cause system crashes and thereby lead to DoS. In addition, Memory Leak can cause reliability problems, Missing Lock Check can cause data modification, Integer Overflow can cause resource consumption, and Unused Variable can cause quality degradation.

Moreover, we gained 19 CVEs, which show the harmfulness of the API misuses detected by `APP-Miner`. One of the CVEs is shown in Figure 5.

It is a NULL Pointer Dereference vulnerability, which can cause a system crash. Specifically, since the memory may be insufficient, the *vzalloc* may fail and return a NULL pointer. However, there was no check for the return value. As a result, the NULL pointer would be used and cause a system crash. The CVSS v3 base score of the vulnerability is 5.5, the medium level of security impact.

5.3. Comparison with State-of-Art Tools

We compared `APP-Miner` with two categories of tools: (1) extracting API patterns from the source code and (2) extracting API patterns from the documentation. We mainly focus on how many API path patterns and API misuses could only be found by `APP-Miner`. Thus, we used the extracted patterns and accepted API misuses found by `APP-Miner` as the benchmark.

```
/* drivers/media/test-drivers/vidtv/
 * vidtv_s302m.c */
@@ -455,6 +455,9 @@ struct vidtv_encoder
     e->name = kstrdup(args.name,
     ↪  GFP_KERNEL);

   e->encoder_buf =
   ↪  vzalloc(VIDTV_S302M_BUF_SZ);
+  if (!e->encoder_buf)
+    goto out_kfree_e;
+
   e->encoder_buf_sz = VIDTV_S302M_BUF_SZ;
   e->encoder_buf_offset = 0;
```

Figure 5: CVE-2022-3078 [8]

TABLE 5: Comparison results of Crix and APISan

Tools	# Patterns	# Violated patterns	# API misuses
APP-Miner	3,985	52	116
Crix	260	14	5
APISan	0	0	0

5.3.1. Comparison with Tools Extracting API Patterns From the Source Code. We compared `APP-Miner` in Linux kernel 5.15-rc7 with two state-of-the-art tools: Crix [4] and APISan [2].

Table 5 shows the comparison results. Few of the violated patterns and the API misuses are found by Crix and APISan. Crix is designed to extract the patterns that the APIs should be used with security checks. However, these types of patterns only account for a small part. Moreover, Crix fails to detect the API misuses whose contexts differ from the API patterns. APISan collects the API usages with the same contexts to extract the API patterns. However, `APP-Miner` extracts the API patterns that frequently appear in different contexts.

Further investigating the experiment results, we found that many similar APIs have the same path patterns. For example, the path patterns of memory allocation APIs are usually $\{$*memory allocation* $\rightarrow$ *check*$\}$. It is such well-known knowledge that APISan's return value checker artificially increases the weight of the APIs whose names contain "alloc". However, not all such APIs' return values should be checked. One of the exceptions is "mempool_alloc", which will not return a NULL pointer and is usually unchecked in the source code. Therefore, `APP-Miner` can provide a precise API list that the return value of which memory allocation APIs should be checked, rather than a rough filter that checks whose names contain "alloc".

5.3.2. Comparison with Tool Extracting API Patterns From the Documentation. Advance [1] is a state-of-the-art tool extracting API patterns from documentation by NLP technologies. Since it extracts API

TABLE 6: Comparison results of Advance

Tools	# Patterns	# Violated patterns	# API misuses
APP-Miner	598	2	3
Advance	35	0	0

patterns from the library and detects API misuses in the software, we implement our tool on the OpenSSL and Apache httpd that use the API path patterns extracted from the OpenSSL to detect the misuses of OpenSSL APIs in Apache httpd.

Table 6 shows the comparison results of Advance. We can find that Advance uncovers most of the patterns extracted by APP-Miner. That is mainly because the documentation always selectively records a part of commonly used APIs.

Moreover, Advance focuses on the records containing "before" or "after" to gain the sequential order of operations. However, the OpenSSL documentation mainly describes the return value but ignores the description of something like "you must check the return value after use". Figure 6 shows a misuse of *ASN1_STRING_new* that violates the pattern $\{ASN1\_STRING\_new \rightarrow check\}$. However, the documentation [3] only records the return value but ignores the return value check: "*ASN1_STRING_new* returns a valid *ASN1_STRING* structure or NULL if an error occurred". Therefore, Advance fails to infer the pattern and cannot detect the misuse.

5.4. False Positives and False Negatives

We extracted the top 10 violated API patterns and the top 5 violations of each pattern as the test cases from APP-Miner, Crix, and APISan. However, (1) One of the patterns was reported by both APP-Miner and Crix, and (2) Crix and APISan had only 26 and 13 violations from the first 10 patterns. Therefore, the test cases contain 29 API patterns and 89 violations. Even

```
/* modules/ssl/ssl_engine_vars.c */
@@ -1104,6 +1104,9 @@
   ASN1_STRING *ret = ASN1_STRING_new();
   int rv = 0;

+  if (!ret)
+    return rv;
+
   if (d2i_DISPLAYTEXT(&ret, &pp,
   ↪  str->length)) {
```

Figure 6: A NULL Pointer Dereference bug in Apache httpd identified by APP-Miner

TABLE 7: The precision and recall evaluation

Tools	Extracted Patterns				Detected Bugs			
	TP	FP	FN	F1	TP	FP	FN	F1
APP-Miner	17	0	4	0.89	29	27	1	0.67
Crix	12	0	9	0.73	7	19	23	0.25
APISan	4	8	17	0.24	0	13	30	-
Documentation	4	-	17	0.32	-	-	-	-

though APP-Miner provided more violations than the other two tools, it was acceptable that the violations could contain more false positives, and the ability to report more violations was indeed the advantage of APP-Miner.

We manually checked the test cases, finding 21 true patterns and 30 true bugs. Specifically, we treated the one as a false pattern if it is unrelated to the code security. For example, APISan reported a false pattern that "fprintf" should be used if "open" is used. We treated the one as a false bug if it did not violate the patterns or it would not cause security problems. After that, we tested the tools shown in Table 7. Furthermore, We compared the extracted patterns with the Linux kernel API documentation [16], the handwritten patterns.

Our tool can find the most TP patterns and bugs. Not only that, but our F1 scores are higher than the others. The reason is that many new patterns extracted by our tool are hardly used to detect the bugs before. Therefore, the possibilities of finding true bugs are much higher.

Moreover, 14 patterns extracted by APP-Miner are uncovered in the documentation. It shows that many patterns extracted by APP-Miner can complement the documentation. Moreover, since Advance extracts the API patterns from the documentation, the results can prove that Advance will fail to gain these 14 patterns.

The leading causes of the false positive bugs reported by APP-Miner are as follows:

- **Inconsistent parameters:** As mentioned in §3.3.2, APP-Miner only focuses on the number of parameters but does not consider the values and the types of parameters. For example, memory allocation APIs may return NULL pointers if its last parameter is "*GFP_KERNEL*". However, they will not return NULL pointers if the last parameter is "*__GFP_NOFAIL*". Consequently, APP-Miner will falsely report the violations that they are missing return value checks. We will discuss further improvements of these false positives in §6.1.
- **Imprecise data flow analysis:** Because of the complex pointer mechanism in C language, it is hard for APP-Miner to gain all the APIs' data-related operations and thereby fail to generate

complete API paths. As a result, APP-Miner may report the incomplete API paths as violations, which should not be. For example, there exist substantial nested structures in Linux kernel, which contain multiple levels of pointers. It is a complex problem to analyze their data flow, even with state-of-the-art static analysis techniques.

- **Paths across functions:** As discussed in §3.2, APP-Miner only generates API paths from each function to avoid complex interprocedural analysis. Unfortunately, some programmers split the API patterns into different functions executed sequentially. As a result, APP-Miner will falsely consider it a violation. For example, APP-Miner reports a misuse of *skb_get_rx_queue* in *netvsc_xdp_xmit* that *skb_record_rx_queue* is missed. However, the caller of *netvsc_xdp_xmit* already used *skb_record_rx_queue* before calling *netvsc_xdp_xmit*. After interacting with the maintainers, they accepted our patch, adding a comment for *netvsc_xdp_xmit* to avoid future API misuses, as shown in Figure 7. An interprocedural analysis is necessary to avoid this false positive. However, employing interprocedural analysis on large software is a heavy task with current computing power. In summary, this type of false positive is acceptable and can warn programmers to avoid future errors.

Furthermore, the false negative bugs reported by APP-Miner are because APP-Miner fails to extract the API patterns. Since Crix and APISan only consider the code pieces in similar contexts, their *confidence_threshold* can be more easily reached than APP-Miner, thereby detecting some API patterns with low frequency. However, it is acceptable because low-frequency API patterns are false-prone.

```
/* drivers/net/hyperv/netvsc_drv.c */
@@ -803,6 +803,7 @@
+/* This function should only be called
↪  after skb_record_rx_queue() */
 static void netvsc_xdp_xmit(struct
 ↪  sk_buff *skb, struct net_device
 ↪  *ndev)
 {
   int rc;

   skb->queue_mapping =
   ↪  skb_get_rx_queue(skb);
```

Figure 7: Patch that adds a comment to avoid future API misuses

TABLE 8: Time and space cost of APP-Miner

Software	Extracting patterns		Detecting violations	
	Time (s)	Space (MB)	Time (s)	Space (MB)
Linux kernel	6,343	1,283	171	876
OpenSSL	3,632	1,894	84	781
FFmpeg	176	829	42	302
Apache httpd	-	-	29	149

In order to evaluate the full set of our API path patterns, we manually checked all the 3,985 extracted API path patterns from Linux kernel. The API path patterns can be classified into nine categories: Memory Allocation/Release, Driver/Device Registration, Enable Operation, Character/Digital Computation, Data Operation, Discriminate Check, File Operation, Initialization/Finalization, and Lock/Unlock.

Moreover, we found 750 false positives, which account for 18.8% of the total. The leading causes of false positives are as follows:

Patterns across functions: Since we generate the API paths intraprocedurally, we fail to gain the complete API path patterns if the operations are separated into different functions. As a result, the extract API path patterns are just something like $\{check \rightarrow memory\ release\}$, which should be $\{memory\ allocation \rightarrow check \rightarrow memory\ release\}$. Similar to **Paths across functions**, it is hard to solve the problem with current computing power.

Functional partner: Some APIs unrelated to the code security often appear together to accomplish the missions. For example, *iter_begin*, *iter_next*, and *iter_end* are used together to traverse lists. These false positives are acceptable since they can guide programmers to use the APIs.

Debug APIs: There are debug APIs that print debug information. When they are used for debugging some APIs, APP-Miner will falsely regard the debug APIs as part of the API path patterns. We will discuss further improvements of these false positives in §6.2.

5.5. Performance

We used 24 threads to run the APP-Miner. The time and memory costs are shown in Table 8. For example, it takes about 1.76 hours to extract 3,985 API path patterns from Linux kernel. The average works out roundly at 2.3 seconds. Moreover, APP-Miner takes a few minutes to detect violations from ten thousand c files in Linux kernel. Besides, the maximum memory used by APP-Miner is less than two GB.

Furthermore, we compared the time with Crix and APISan when employed in Linux kernel, which are

5,290 seconds and 1,862 seconds, respectively. We can find no order of magnitude difference. The time comparison results prove that APP−Miner has the scalability to be employed in large software.

6. Discussion & Future Work

While APP−Miner is very effective in automatically extracting API path patterns and detecting violations, our current version of APP−Miner has the following limitations, which will be addressed in our future work.

6.1. Inconsistent Parameters

As mentioned in §5.4, inconsistent parameters will cause false positives reported by APP−Miner. In order to solve the problem, we will consider the values and the types of parameters as the third label of the nodes in API paths. As a result, APP−Miner can distinguish the same API with different parameters and extract respective path patterns to avoid this false positive. For example, the path pattern of *kmalloc(GFP_KERNEL)* has the return value check, while the path pattern of *kmalloc(__GFP_NOFAIL)* does not.

6.2. Debug APIs

Many debug APIs appear frequently but are unrelated to the code security. In the future, we can put different weights on the APIs' data-related operations that decrease the effect of the debug APIs.

6.3. Equivalent Functions

There are some equivalent functions with different names. APP−Miner will treat them as different functions. Therefore, APP−Miner will report false positives if the equivalent ones replace the functions in the API path patterns. To solve the problem, we can use clustering technologies to classify similar functions into one class and regard them as the same node in the API paths.

7. Related Work

7.1. Extracting API Patterns From the Source Code

Much work tends to extract API patterns from the source code. Crix [35] searches for paired APIs and checks. PR-Miner [32] uses frequent itemset mining to extract implicit program rules. Doc2Spec [54] and Advance [36] analyze the API documentation by NLP technologies to infer API patterns. APISan [52] uses the statistical method to extract API patterns. NAR-Miner [21] utilizes frequent itemset mining to discover negative program rules and detect mutex operations. JUXTA [37] extracts several API patterns for file system access. Bugram [48] utilizes an n-gram model to infer the most probably correct sequence of the operations. Dawson [24] extracts implicit program rules to detect missing checks.

7.2. Frequent Subgraph Mining

ScaleMine [17] uses an approximate phase to extract frequent subgraphs in a large graph. AGM [26] generates candidate subgraphs from one to k nodes and computes their support. FSG [29] combines (k-1)-node frequent subgraphs to build k-node candidate subgraphs and filter the infrequent candidates. GSpan [51] extracts neighborhoods of subgraphs and computes their frequency. FFSM [25] employs a vertical search scheme to decrease redundant candidates.

8. Conclusion

Violating API patterns will cause API misuses and lead to various security issues. This paper presents APP−Miner, a general framework that uses the frequent subgraph mining technique to automatically extract API path patterns to detect API misuses without predefined templates. According to the challenges, we use a new method to correctly and efficiently extract API path patterns from the source code.

We have evaluated APP−Miner with four widely used open-source software, including Linux kernel, OpenSSL, FFmpeg, and Apache httpd. APP−Miner extracted 4,788 API path patterns from the first three. In addition, APP−Miner has detected many violations using the extracted API path patterns. As a result, we found 116 API misuses in the master branches of Linux kernel, 35 of OpenSSL, 3 of FFmpeg, and 3 of Apache httpd. Moreover, we gained 19 CVEs.

Acknowledgment

We thank the anonymous reviewers for their constructive feedback and encouragement. This paper is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No.XDA01020304, the National Natural Science Foundation of China under No.62202457, and the project funded by China Postdoctoral Science Foundation under No.2022M713253. Data and experiments conducted in this paper are supported by Open Source Map Large Research Infrastructure.

References

[1] Advance. [Online]. Available: https://github.com/lvtao-sec/Advance

[2] APISan: Sanitizing API usages through semantic cross-checking. [Online]. Available: https://github.com/sslab-gatech/apisan/

[3] ASN1_STRING_new. [Online]. Available: https://www.openssl.org/docs/manmaster/man3/ASN1_STRING_new.html

[4] Crix: Detecting missing-check bugs in OS kernels. [Online]. Available: https://github.com/umnsec/crix/

[5] CVE-2021-33909. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33909

[6] CVE-2021-43267. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-43267

[7] CVE-2021-45480. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45480

[8] CVE-2022-3078. [Online]. Available: https://access.redhat.com/security/cve/CVE-2022-3078

[9] Deadline. [Online]. Available: https://github.com/sslab-gatech/deadline

[10] LLVMGetNextBasicBlock. [Online]. Available: https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html#ga07967d9c5eb0aa3dd8c2f0a0068ae3aa

[11] LLVMGetNextInstruction. [Online]. Available: https://llvm.org/doxygen/group__LLVMCCoreValueInstruction.html#ga1b4c3bd197e86e8bffdda247ddf8ec5e

[12] LLVMGetOperand. [Online]. Available: https://llvm.org/doxygen/group__LLVMCCoreValueUser.html#ga799d58a361054323cb457945071cbfdb

[13] LLVMGetUser. [Online]. Available: https://llvm.org/doxygen/group__LLVMCCoreValueUses.html#ga24f4b24c04a81ad75566021043f91848

[14] Proof by contrapositive. [Online]. Available: https://en.wikipedia.org/wiki/Proof_by_contrapositive

[15] Search CVE List. [Online]. Available: https://cve.mitre.org/cve/search_cve_list.html

[16] The Linux Kernel API. [Online]. Available: https://docs.kernel.org/core-api/kernel-api.html

[17] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. T. Jamour, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 716–727.

[18] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 25–34.

[19] M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, "Finding bugs using your own code: Detecting functionally-similar yet inconsistent code," in *30th USENIX Security Symposium*, 2021, pp. 2025–2040.

[20] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 211–221.

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

[22] R. Chang, A. Podgurski, and J. Yang, "Finding what's not there: a new approach to revealing neglected conditions in software," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, 2007, pp. 163–173.

[23] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining," *IEEE Trans. Software Eng.*, pp. 243–257, 2012.

[24] E. Dawson, C. D. Yu, H. Seth, C. Andy, and C. Benjamin, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001, p. 57–72.

[25] J. Huan, W. Wang, and J. F. Prins, "Efficient mining of frequent subgraphs in the presence of isomorphism," in *Proceedings of the 3rd IEEE International Conference on Data Mining*, 2003, pp. 549–552.

[26] A. Inokuchi, T. Washio, and H. Motoda, "Complete mining of frequent patterns from graphs: Mining graph data," *Mach. Learn.*, pp. 321–354, 2003.

[27] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *29th USENIX Security Symposium*, 2020, pp. 2271–2287.

[28] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 178–189.

[29] M. Kuramochi and G. Karypis, "An efficient algorithm for discovering frequent subgraphs," *IEEE Trans. Knowl. Data Eng.*, pp. 1038–1051, 2004.

[30] C. Lee, F. Chen, and G. Rosu, "Mining parametric specifications," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 591–600.

[31] C. Lemieux, "Mining temporal properties of data invariants," in *37th IEEE/ACM International Conference on Software Engineering*, 2015, pp. 751–753.

[32] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 306–315.

[33] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "AntMiner: mining more bugs by reducing noise interference," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 333–344.

[34] V. B. Livshits and T. Zimmermann, "DynaMine: Finding common error patterns by mining software revision histories," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 296–305.

[35] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences," in *28th USENIX Security Symposium*, 2019, pp. 1769–1786.

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

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

[38] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 166–177.

[39] A. Pakki and K. Lu, "Exaggerated error handling hurts! an in-depth study and context-aware detection," in *2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1203–1218.

[40] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE 2009*, 2009, pp. 371–382.

[41] ——, "Leveraging test generation and specification mining for automated bug detection without false positives," in *34th International Conference on Software Engineering*, 2012, pp. 288–298.

[42] G. Reger, H. Barringer, and D. E. Rydeheard, "A pattern-based approach to parametric specification mining," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 658–663.

[43] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Software Eng.*, pp. 613–637, 2013.

[44] S. Saha, J. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013, pp. 1–12.

[45] B. Sun, G. Shu, A. Podgurski, and B. Robinson, "Extending static analysis by mining project-specific rules," in *34th International Conference on Software Engineering*, 2012, pp. 1054–1063.

[46] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *31st International Conference on Software Engineering*, 2009, pp. 496–506.

[47] ——, "Alattin: Mining alternative patterns for defect detection," *Autom. Softw. Eng.*, pp. 293–323, 2011.

[48] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 708–719.

[49] W. Wang, K. Lu, and P. Yew, "Check it again: Detecting lacking-recheck bugs in OS kernels," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1899–1913.

[50] T. Xie and J. Pei, "MAPO: mining API usages from open source repositories," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, pp. 54–57.

[51] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, 2002, pp. 721–724.

[52] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API usages through semantic cross-checking," in *25th USENIX Security Symposium*, 2016, pp. 363–378.

[53] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "PeX: A permission check analysis framework for linux kernel," in *28th USENIX Security Symposium*, 2019, pp. 1205–1220.

[54] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *24th IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 307–318.

Appendix A.
List of Accepted Patches

TABLE 9: List of patches (1 - 55) accepted by the Linux kernel community

Subsystem	Risky API	Impact	Commit ID	CVE
acpi	devres_alloc	System crash	2cea3ec5b009	
atm	dma_map_single	System crash	0f74b29a4f53	
bluetooth	platform_get_irq	System crash	b38cd3b42fba	
bluetooth	platform_driver_register	System crash	ab2d2a982ff7	
clk	kcalloc	System crash	ed713e2bc093	CVE-2022-3114
crypto	alloc_page	System crash	5f21d7d283dd	
dma	dma_set_mask_and_coherent	System crash	2d21543efe33	
fs	kmemdup	System crash	e6c3cef24cb0	
fs	kstrdup	System crash	d97038d5ec20	
fsi	ida_simple_get	System crash	35af9fb49bc5	
gpio	platform_get_irq	System crash	c1bcb976d8fe	
gpu	kzalloc	System crash	fea3fdf975dd	CVE-2022-30853
gpu	kzalloc	System crash	73c3ed7495c6	CVE-2022-3115
gpu	kmemdup	System crash	8027a9ad9b35	
gpu	dma_set_mask_and_coherent	System crash	44ab30b05614	
gpu	dma_set_mask_and_coherent	System crash	4a39156166b9	
gpu	mipi_dsi_driver_register	System crash	831463667b5f	
gpu	kzalloc	System crash	93340e10b9c5	CVE-2023-3220
gpu	kzalloc	System crash	13fcfcb2a9a4	
gpu	kzalloc	System crash	c96988b7d993	
gpu	alloc_ordered_workqueue	System crash	115906ca7b53	
gpu	kmalloc	System crash	d839f0811a31	CVE-2023-3355
gpu	alloc_ordered_workqueue	Reliability	643b7d0869cc	
gpu	alloc_ordered_workqueue	System crash	afe4cb96153a	
hid	devm_kzalloc	System crash	13251ce1dd9b	
hid	kcalloc	System crash	b3d40c3ec3dc	CVE-2023-3358
hid	dma_alloc_coherent	System crash	53ffa6a9f83b	CVE-2023-3357
hwtracing	dma_alloc_coherent	System crash	82f76a4a7207	
i2c	platform_driver_register	System crash	6ba12b56b9b8	
iio	devm_request_threaded_irq	System crash	b30537a4cedc	
infiniband	kmalloc_array	System crash	7694a7de22c5	CVE-2022-3105
iommu	dma_set_mask_and_coherent	System crash	1fdbbfd5099f	
md	alloc_percpu	System crash	d3aa3e060c4a	
media	vzalloc	Reliability	e6a21a14106d	CVE-2022-3078
media	devm_kzalloc	System crash	e25a89f743b1	CVE-2022-3113
media	dma_set_coherent_mask	System crash	43f0633f8994	
media	kmalloc	System crash	6e5e5defdb8b	
media	coda_iram_alloc	System crash	6b8082238fb8	
media	kmalloc	System crash	f30ce3d3760b	
media	devm_regulator_get	System crash	da8e05f84a11	
media	create_workqueue	System crash	2371adeab717	
memory	setup_interrupts	System crash	fd7bd80b4637	
memstick	alloc_ordered_workqueue	Reliability	4f431a047a5c	
mfd	mc13xxx_irq_request	System crash	e477e51a41cb	
mfd	platform_driver_register	System crash	8325a6c24ad7	
mfd	platform_get_resource	System crash	d918e0d58244	
misc	kmalloc	System crash	4a9800c81d2f	CVE-2022-3104
misc	pci_enable_device	System crash	9c27896ac1bb	
mmc	dma_set_mask_and_coherent	System crash	40c67c291a93	
mmc	platform_get_resource	System crash	4d315357b3d6	
mmc	clk_enable	System crash	09e7af76db02	
mtd	platform_get_irq	System crash	3e68f331c8c7	
net	kmalloc	System crash	407ecd1bd726	CVE-2022-3106
net	kvmalloc_array	System crash	886e44c9298a	CVE-2022-3107
net	kmemdup	System crash	abfaf0eee979	CVE-2022-3108

TABLE 10: List of patches (56 - 116) accepted by the Linux kernel community

Subsystem	Risky API	Impact	Commit ID	CVE
net	dma_set_coherent_mask	System crash	128f6ec95a28	
net	devm_ioremap	System crash	d5a73ec96cc5	
net	_usecs_to_jiffies	Resource consumption	acde891c243c	
net	kcalloc	System crash	60ec7fcfe768	
net	platform_get_irq	System crash	db6d6afe382d	
net	platform_get_irq	System crash	cb93b3e11d40	
net	platform_get_irq	System crash	99d7fbb5cedf	
net	kcalloc	System crash	bdf1b5c3884f	
net	kcalloc	System crash	9b8bdd1eb589	
net	nla_put_u32	System crash	92a34ab169f9	
net	platform_get_irq	System crash	c6564c13dae2	
net	ioremap	System crash	7e4760713391	
net	kmemdup	System crash	a72c01a94f1d	
net	nla_memdup	System crash	6ad27f522cb3	
net	clk_enable	System crash	6babfc6e6fab	
net	clk_enable	System crash	2169b79258c8	
net	pfkey_broadcast	System crash	4dc2a5a8f675	
net	devm_clk_get	System crash	68b4f9e0bdd0	
net	devm_kcalloc	System crash	cd07eadd5147	
net	create_singlethread_workqueue	System crash	26e6775f7551	
net	create_singlethread_workqueue	System crash	1fdeb8b9f29d	
ntb	devm_kcalloc	System crash	2790143f0993	
power	platform_get_resource	System crash	1c1348bf056d	
power	wm8350_register_irq	System crash	b0b14b5ba11b	
rtc	wm8350_register_irq	System crash	43f0269b6b89	
scsi	ioremap	System crash	2576e153cd98	
scsi	dma_alloc_coherent	System crash	aa7069d840da	
scsi	dma_map_single	System crash	32fe45274edb	
soc	ioremap	System crash	a222fd854139	
soc	devm_clk_get	System crash	9de2b9286a6d	
soc	devm_kcalloc	System crash	5a811126d38f	
soc	idr_alloc	System crash	6d7860f5750d	
sound	of_get_child_by_name	System crash	f7a6021aaf02	
sound	devm_regmap_init_mmio	System crash	aa505ecccf2a	
sound	device_property_read_u32_array	System crash	2167c0b20596	
sound	ioremap	System crash	3ecb46755eb8	
sound	dma_set_mask_and_coherent	System crash	1b1f98dd70dc	
sound	clk_enable	System crash	ca1697eb0920	
sound	clk_enable	System crash	ed7c9fef1193	
sound	clk_enable	System crash	2ecf362d2203	
sound	clk_enable	System crash	45ea97d74313	
sound	clk_enable	System crash	f9e2ca0640e5	
sound	snd_soc_dai_stream_valid	System crash	de2c6f98817f	
sound	clk_enable	System crash	405afed8a728	
sound	wm8350_register_irq	System crash	db0350da8084	
sound	i2c_add_driver	System crash	82fa8f581a95	
sound	rsnd_mod_power_on	System crash	376be51caf88	
sound	copy_to_user	System crash	d067b3378a78	
sound	devm_kcalloc	System crash	60591bbf6d5e	
spi	dma_set_mask	System crash	13262fc26c18	
staging	rtw_alloc_hwxmits	System crash	f94b47c6bde6	CVE-2022-3110
staging	amvdec_add_ts	System crash	c8c80c996182	CVE-2022-3112
staging	devm_kcalloc	System crash	2e81948177d7	
thermal	kmemdup	System crash	38b16d6cfe54	
uio	dma_set_coherent_mask	System crash	eec91694f927	
usb	device_create_file	System crash	0f6632e2e8be	
usb	kcalloc	System crash	c35ca10f53c5	
video	platform_get_resource	System crash	9d54c5d47406	
video	device_create_file	System crash	e69dade8a4cf	
virt	device_create_file	System crash	d4d2c58bdb91	
watchdog	platform_driver_register	System crash	97d5ec548150	

TABLE 11: List of patches (1 - 34) accepted by the OpenSSL community

Subsystem	Risky API	Impact	Commit ID	CVE
apps	OPENSSL_strdup	System crash	0c5905581e9d	
apps	OPENSSL_strdup	System crash	79cda38cff83	
apps	OPENSSL_strdup	System crash	8f084b43803d	
apps	dup_bio_out	System crash	ba0b60c632ae	
apps	OPENSSL_strdup	System crash	a6a2dd9f60b3	
crypto	X509_STORE_lock	Data modification	814999cb4413	
crypto	BN_BLINDING_lock	Data modification	aefbcde29166	
crypto	get_globals	System crash	7f1cb465c1f0	
crypto	OPENSSL_memdup	System crash	3f6a12a07f52	
crypto	rand_get_global	System crash	09dca557332a	
crypto	OPENSSL_strndup	System crash	366a16263959	
crypto	BIO_read	System crash	2823e2e1d394	
crypto	OPENSSL_strdup	System crash	816d6e578ccc	
crypto	OPENSSL_strdup	System crash	e163969d3580	
crypto	BN_CTX_start	Reliability	050dddb06162	
fuzz	BIO_new	System crash	d43597c718dd	
fuzz	OSSL_LIB_CTX_new	System crash	885d97fbf84f	
fuzz	ASN1_item_i2d	System crash	1cb35ce06a96	
providers	OPENSSL_strdup	System crash	c920020f0bb1	
ssl	BN_dup	System crash	12e488367d34	
test	OPENSSL_zalloc	System crash	2208ba56ebef	
test	OPENSSL_strdup	System crash	b2f90e93a07d	
test	sk_SCT_new_null	System crash	7625d70ad9e7	
test	OPENSSL_strdup	System crash	09030ee73693	
test	OPENSSL_strdup	System crash	17da5f2af833	
test	X509_STORE_CTX_free	Quality degradation	f541419c7926	
test	BIO_set_conn_ip_family	System crash	8c590a219fe3	
test	OSSL_PROVIDER_load	System crash	78c5f1266fdd	
test	glue2bio	System crash	18cb1740cc0f	
test	BIO_new_mem_buf	System crash	cf21d1c62dcd	
test	SSL_CTX_new	System crash	b0317df23117	
test	OPENSSL_malloc	System crash	b2feb9f0e394	
test	OPENSSL_zalloc	System crash	4f4942a133bd	
test	OPENSSL_malloc	System crash	b147b9daf177	
test	OPENSSL_strdup	System crash	5203a8dfdc20	

TABLE 12: List of patches (1 - 3) accepted by the FFmpeg community

Subsystem	Risky API	Impact	Commit ID	CVE
libavcodec	av_malloc	System crash	656cb0450aeb	CVE-2022-3109
libavcodec	av_mallocz	System crash	c4d63dbc9417	CVE-2022-30856
libavformat	avformat_new_stream	System crash	9cf652cef49d	CVE-2022-3341

TABLE 13: List of patches (1 - 3) accepted by the Apache httpd community

Subsystem	Risky API	Impact	Commit ID	CVE
ssl	ASN1_STRING_new	System crash	r1898367	
ssl	X509_STORE_CTX_init	System crash	r1898368	
ssl	X509_STORE_CTX_init	System crash	r1898410	

Appendix B.
Meta-Review

B.1. Summary

The paper presents a new analysis framework `APP-Miner` for detecting API misuses. `APP-Miner` relies on static analysis to extract common API path patterns via frequent subgraph mining. `APP-Miner` discovered a wide range of API misuses.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field
- Identifies an Impactful Vulnerability

B.3. Reasons for Acceptance

1) Creates a New Tool to Enable Future Science: The paper presents a new analysis framework `APP-Miner` for detecting API misuses. The authors note challenges that could hinder extracting accurate and complete API path patterns and propose new methods to address the challenges. The tool conducts topologizing, completion, and index building, all facilitating extracting API path patterns efficiently.
2) Provides a Valuable Step Forward in an Established Field: `APP-Miner` provides a new angle to abstract API usages and then check existing source code for potential violations of the mined patterns.
3) Identifies an Impactful Vulnerability: `APP-Miner` discovers a wide range of new API misuses, leading to a total of 19 CVEs in production.