Detecting Functionality-Specific Vulnerabilities via Retrieving Individual Functionality-Equivalent APIs in Open-Source Repositories

Tianyu Chen
Peking University, Beijing, China

Zeyu Wang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Lin Li
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Ding Li
Peking University, Beijing, China

Zongyang Li
Peking University, Beijing, China

Xiaoning Chang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Pan Bian
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Guangtai Liang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Qianxiang Wang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Tao Xie[1]
Peking University, Beijing, China

Abstract

Functionality-specific vulnerabilities, which mainly occur in Application Programming Interfaces (APIs) with specific functionalities, are crucial for software developers to detect and avoid. When detecting individual functionality-specific vulnerabilities, the existing two categories of approaches are ineffective because they consider only the API bodies and are unable to handle diverse implementations of functionality-equivalent APIs. To effectively detect functionality-specific vulnerabilities, we propose APISS, the first approach to utilize API doc strings and signatures instead of API bodies. APISS first retrieves functionality-equivalent APIs for APIs with existing vulnerabilities and then migrates Proof-of-Concepts (PoCs) of the existing vulnerabilities for newly detected vulnerable APIs. To retrieve functionality-equivalent APIs, we leverage a Large Language Model for API embedding to improve the accuracy and address the effectiveness and scalability issues suffered by the existing approaches. To migrate PoCs of the existing vulnerabilities for newly detected vulnerable APIs, we design a semi-automatic schema to substantially reduce manual costs. We conduct a comprehensive evaluation to empirically compare APISS with four state-of-the-art approaches of detecting vulnerabilities and two state-of-the-art approaches of retrieving functionality-equivalent APIs. The evaluation subjects include 180 widely used Java repositories using 10 existing vulnerabilities, along with their PoCs. The results show that APISS effectively retrieves functionality-equivalent APIs, achieving a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55. APISS is highly efficient: the manual costs are within 10 minutes per vulnerability and the end-to-end runtime overhead of testing one candidate API is less than 2 hours. APISS detects 179 new vulnerabilities and receives 60 new CVE IDs, bringing high value to security practice.

2012 ACM Subject Classification Security and privacy → Software security engineering

Keywords and phrases Application Security, Vulnerability Detection, Large Language Model

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.6

[1] Corresponding author

© Tianyu Chen, Zeyu Wang, Lin Li, Ding Li, Zongyang Li, Xiaoning Chang, Pan Bian, Guangtai Liang, Qianxiang Wang, and Tao Xie;
licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 6; pp. 6:1–6:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work was partially supported by National Natural Science Foundation of China under Grant No. 92464301. Tao Xie is also affiliated with the School of Computer Science, Peking University; Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China.

1 Introduction

Application Programming Interfaces (APIs) in open source software are critical and widely used in software development [40], while the usage of APIs also increases the burden on developers to avoid security vulnerabilities among the APIs. Functionality-specific vulnerabilities, as one common type of API vulnerabilities, are defined as those that mainly occur in APIs with specific functionalities. For example, `Path Traversal` vulnerabilities (i.e., unauthorized access to a restricted directory) likely occur in an `unzip` API if it forgets to check whether the paths in its inputs (an uploaded zip file) are permitted. These vulnerabilities are often caused by lacking crucial operations during implementation, such as input validation, so similar vulnerabilities might occur among individual APIs with equivalent/similar functionalities (referred to as functionality-equivalent APIs).

However, the existing approaches, which fall into two categories, are ineffective in detecting functionality-specific vulnerabilities, as these approaches consider only the API bodies. First, some approaches [43, 26] aim to detect recurring vulnerabilities, which come from reused code base or shared code logic. However, these approaches find only vulnerabilities that have both similar functionalities and similar implementations with existing ones, thus suffering from high false negatives when encountering functionality-equivalent APIs with diverse implementations. Second, some other approaches [41, 52, 39, 8, 38, 37, 42, 31, 17, 7] aim to detect vulnerabilities other than recurring ones, typically via deep learning, which learns code features of vulnerable APIs/methods of existing vulnerabilities. However, these approaches are ineffective because the APIs affected by these vulnerabilities have diverse implementations without a common pattern. Additionally, these approaches cannot provide Proof-of-Concepts (PoCs) for the detected vulnerabilities, leading to high manual costs in vulnerability validation.

To effectively detect functionality-specific vulnerabilities, we propose APISS, the first approach that utilizes API doc strings and signatures (instead of focusing on API bodies), consisting of two steps: retrieving functionality-equivalent APIs for existing vulnerabilities and then migrating PoCs of the existing vulnerabilities to exploit new vulnerabilities. Our rationale comes from our observation of functionality-equivalent APIs: these APIs have similar doc strings and signatures because an API's input/output format (i.e., the API name, parameters, and output type included in its signature) is determined by this API's functionality and the doc string is used to explain each component of this API's signature. Thus, the doc strings and signatures of APIs with existing vulnerabilities can be utilized to retrieve functionality-equivalent APIs with potential functionality-specific vulnerabilities.

There are two main challenges in the two steps of APISS, respectively. First, the main challenge of retrieving functionality-equivalent APIs is the effectiveness and scalability issues. The effectiveness issue of existing approaches [29, 11, 6, 47] is caused by their overemphasis on the similarities of API bodies, not being able to handle diverse implementations of functionality-equivalent APIs. The scalability issue of these existing approaches comes from their training/fine-tuning on specific datasets, e.g., mapping a Java API to a C# API [29, 11, 6] or a Swift one [47]. Second, the main challenge of migrating the PoCs of existing vulnerabilities is the high manual costs. Migrating the PoCs requires building the environment and API dependencies of the project repository that includes a newly-detected

vulnerable API. Given that newly-detected vulnerable APIs come from different repositories and have different context dependencies, environment/dependency building incurs high manual costs.

To address the preceding two challenges, we design two respective techniques based on our observations on functionality-equivalent APIs and PoCs of functionality-specific vulnerabilities. First, we leverage a Large Language Model (LLM) for API embedding to address the effectiveness and scalability issues due to the enormous achievements of LLMs. For each pair of APIs, we combine the embedding distances of each component (doc string, signature, body) between them to determine whether they are functionality-equivalent. Our observation is the similarities among API doc strings and signatures of functionality-equivalent APIs. Second, we design a semi-automatic schema to migrate the PoC of existing vulnerabilities, thus substantially reducing manual costs. Our observation is that a vulnerability PoC consists of two parts: input construction (constructing a specific object) and API invocation (invoking the target API) where the former remains the same during migration and the latter often accounts for one or two lines of PoCs and can be semi-automatically migrated by the algorithm of parameter mapping [45].

We evaluate APISS on 180 widely used Java repositories using 10 existing vulnerabilities along with their PoCs. Additionally, we compare APISS's effectiveness with four state-of-the-art approaches of detecting vulnerabilities and two state-of-the-art approaches of retrieving functionality-equivalent APIs. Our evaluation results show that APISS is effective in detecting new vulnerabilities as the best of the baselines under comparison can detect only 58% of the vulnerabilities detected by APISS. Additionally, APISS effectively retrieves functionality-equivalent APIs, achieving a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55. APISS is highly efficient: its manual costs are within 10 minutes per vulnerability and the end-to-end runtime overhead of analyzing one candidate API is less than 2 hours. APISS has detected 170 new vulnerabilities and received 60 new CVE IDs, bringing high value to security practice.

In summary, this paper makes the following main contributions:

- To the best of our knowledge, we are the first to utilize API doc strings and signatures instead of focusing on API bodies to retrieve functionality-equivalent APIs and then detect functionality-specific vulnerabilities.
- We propose an effective technique of API mapping to retrieve functionality-equivalent APIs mainly based on the embedding distance of API doc strings and signatures, and achieve a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55.
- We design a semi-automatic schema to reduce the manual costs of migrating the PoC of existing vulnerabilities within 10 minutes per vulnerability.
- We implement a prototype of APISS and show its high value to security practice by detecting 179 new vulnerabilities and receiving 60 new CVE IDs.

2 Preliminaries

In this section, we provide definitions for three key concepts to improve accessibility for a broader audience.

CVE (Common Vulnerabilities and Exposures): It is maintained by the MITRE Corporation. CVEs are used to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. Once a vulnerability is discovered, developers can request a unique CVE ID from MITRE. Once assigned, this ID enables consistent reference to the vulnerability across databases and platforms. CVEs are widely used by security practitioners to monitor relevant vulnerabilities and track vulnerability patches and affected packages.

CWE (Common Weakness Enumeration): It is also maintained by MITRE. CWE provides a classification system for types of software vulnerabilities. As of now, there are 943 unique CWE types, offering a standardized framework for understanding software vulnerabilities.

PoC (Proof-of-Concept): A PoC is a minimal, verifiable demonstration that confirms the existence and exploitability of a specific security vulnerability. In our paper, we define a PoC as **a single unit test case that demonstrates such exploitability**. Due to the potential security risks, PoCs are often withheld by practitioners to prevent attackers from exploiting vulnerabilities before affected users can apply necessary fixes.

3 Motivation

In this section, we use a motivating example to introduce functionality-specific vulnerabilities and functionality-equivalent APIs. We further leverage this example to demonstrate why existing approaches fail to identify these vulnerabilities. Then we explain our observations on functionality-equivalent APIs and our insights of APISS.

3.1 A Motivating Example

Listing 1 and Listing 2 show a motivating example of two functionality-equivalent APIs with similar functionality-specific vulnerabilities, CVE-2023-27603[1] and CVE-2023-33484[2]. Specifically, CVE-2023-27603 is an existing Scala vulnerability whose PoC is available while CVE-2023-33484 is a Java vulnerability newly detected by APISS based on the former vulnerability.

Functionality-Specific Vulnerabilities. Functionality-specific vulnerabilities are defined as vulnerabilities that mainly occur in APIs with specific functionalities. These vulnerabilities are often caused by a lack of crucial operations during implementation, such as input validation. Listing 1 is the patch of an example functionality-specific vulnerability, CVE-2023-27603, which affects the `fileToUnzip` API in a widely used third-party repository, `Apache Linkis`. This vulnerability corresponds to a representative type of functionality-specific vulnerability, CWE-22 (i.e., improper limitation of a pathname to a restricted directory). Lines 16-21 in Listing 1 of this patch explain how this vulnerability is exploited and fixed. Specifically, the vulnerable API `fileToUnzip` forgets to check whether the file paths (`filePath` in Line 16) in the input zip file (`zipFilePath`) contain improper relative paths, e.g., “`..`”, which has the risk of unzipping into a restricted directory (in Line 23).

Functionality-Equivalent APIs. Functionality-equivalent APIs refer to a collection of APIs that provide equivalent/similar functionalities, such as “unzip a zip file” or “parse a JSON file”. Take Listing 1 and Listing 2 as examples. These two APIs, which come from two repositories (`Apache Linkis` and `Light4J`), respectively, perform the same functionality: unzipping an input zip file to a specific path.

Based on the preceding definitions, functionality-equivalent APIs often share similar or even the same functionality-specific vulnerabilities. For example, the `unzip` API in Listing 2 is also vulnerable to a path traversal vulnerability, CVE-2023-33484, and is fixed by invoking the `validateZipOut` method before unzipping. However, the implementations of these APIs might diverge substantially. For example, `fileToUnzip`’s source code spans only 27 lines while that of `unzip` spans 46 lines.

[1] `https://nvd.nist.gov/vuln/detail/CVE-2023-27603`
[2] `https://github.com/networknt/light-4j/issues/1760`

3.2 Limitations of Existing Approaches

Recurring-Vulnerability Detection Approaches. Some existing approaches [43, 26, 36] are designed to detect recurring vulnerabilities, which come from reused code or shared code logic. For example, MVP [43] utilizes hash values based on the code property graphs of known vulnerable APIs to match similar APIs in open-source repositories. However, these approaches rely heavily on matching the preceding implementation-based features. This reliance may produce high false negatives in detecting functionality-specific vulnerabilities while the affected APIs have diverse implementations.

Deep-Learning Approaches. Some other approaches [41, 52, 39, 8, 38, 37, 42] learn code features extracted from vulnerable APIs/methods including existing vulnerabilities. For example, Devign [52] uses a Graph Neural Network (GNN) to learn the code property graphs (e.g., control/data-flow graphs) of vulnerable APIs. These approaches can effectively detect vulnerabilities caused by incorrect implementation, such as use-after-free, because the source code of their affected APIs has specific patterns, e.g., a pointer operation after a `free` API invocation. However, these approaches are ineffective in detecting functionality-specific vulnerabilities because their affected APIs have diverse implementations without a common pattern. Additionally, these existing approaches also face limitations in generating PoCs for the detected vulnerabilities. A recent study [46] shows that deep-learning models, including LLMs, struggle to generate inputs to cover a specific code branch, let alone generate inputs to exploit a specific vulnerability.

3.3 Observation of Functionality-Equivalent APIs

Considering that an API's source code comprises not only its implementation (i.e., its body) but also a doc string and a signature, we explore the similarities and disparities among these elements in the APIs affected by existing functionality-specific vulnerabilities.

For example, in Listing 1 and Listing 2, we present a pair of functionality-equivalent APIs. Notably, while both APIs share comparable doc strings and signatures, their bodies diverge. Specifically, Line 7 in Listing 1 and Line 8 in Listing 2 represent their signatures. These signatures directly point out their intended functionality of `unzip` and include similar input parameters, such as the zip file's path/name and the destination directory for extraction. Moreover, their doc strings (Lines 1-6 in Listing 1 and Lines 1-7 in Listing 2) explain the purpose of these parameters, thus helping us classify them as functionality-equivalent APIs. However, in terms of their bodies (Lines 8-32 in Listing 1 and Lines 9-19 in Listing 2), there are noticeable disparities, e.g., their differing lengths (27 lines in the former and 46 lines in the latter).

3.4 Our Insight

Our observation reveals that functionality-equivalent APIs often share similarities in their doc strings and signatures, and disparities in their implementations (bodies). Based on this observation, in this paper, we propose to utilize API doc strings and signatures to retrieve functionality-equivalent APIs, thus contributing to detecting functionality-specific vulnerabilities.

A main characteristic of doc strings and signatures is that they are closer to natural language than code. Doc strings are written in natural language and serve the purpose of explaining API functionalities and the purpose of each parameter. Signatures are part of the API source code, primarily consisting of API names and parameters without structured information, e.g., control/data-flow dependencies.

Listing 1 The Vulnerability Patch of CVE-2023-27603 (an Existing Scala Vulnerability).[3]

```
1  /**
2   * @param zipFilePath
3   *   Path of the zip file to be extracted
4   * @param unzipDir
5   *   Path of the destination directory
6   */
7  def fileToUnzip(zipFilePath: String, unzipDir: String): Unit = {
8      val zipIn = new ZipInputStream(new FileInputStream(zipFilePath))
9      Utils.tryFinally {
10     val destDir = new File(unzipDir)
11     if (!destDir.exists()) {
12         destDir.mkdir()
13     }
14     var entry = zipIn.getNextEntry
15     while (entry != null) {
16 -       val filePath = destDir.getPath + File.separator + entry.
       getName
17 +       var entryName = entry.getName
18 +       if (entryName.contains("..") + File.separator)) {
19          throw new IOException("Zip entry contains illegal characters:
             " + entryName)
20 +       }
21 +       val filePath = destDir.getPath + File.separator + entryName
22          if (!entry.isDirectory) {
23            extractFile(zipIn, filePath)
24          } else {
25            val dir = new File(filePath)
26            dir.mkdir
27          }
28          zipIn.closeEntry()
29          entry = zipIn.getNextEntry
30        }
31     }(IOUtils.closeQuietly(zipIn))
32 }
```

Considering the preceding characteristic of doc strings and signatures, we propose to leverage a Large Language Model (LLM) to retrieve functionality-equivalent APIs due to an LLM's significant advancements in natural language and code comprehension. Our proposed approach involves separately embedding each combination of elements (doc string, signature, and body) of a given API as its embedding features. For each API pair, we compute the distances between their respective embedding features and average them as their weighted distances. Notably, our empirical results in Section 5.4 indicate that the embedding features with API bodies have a negligible contribution.

Advantages Over Existing API Mapping Approaches. Our proposed approach has two main advantages when compared to existing work [29, 11, 6, 50, 47], which models the task of retrieving functionality-equivalent APIs as an API mapping task. First, existing

[3] https://github.com/apache/linkis/pull/4279/commits/b3a16c4f1d4ab66d412b0d55bd8c0c35b48775b5

Listing 2 The Vulnerability Patch of CVE-2023-33484 (a Newly-Detected Java Vulnerability).[4]

```
1  /**
2   * Unzips the specified zip file to the specified destination
      directory.
3   * Replaces any files in the destination, if they already exist.
4   * @param zipFilename the name of the zip file to extract
5   * @param destDirname the directory to unzip to
6   * @throws IOException IOException
7   */
8  public static void unzip(String zipFilename, String destDirname) {
9  ... //17 lines
10     final Path destFile = Paths.get(destDir.toString(), file.toString
          ());
11
12 +   // validate unzip cross file
13 +   validateZipOutputFile(zipFilename, new File(destDirname,file.
      toString()), new File(destDirname));
14
15     if(logger.isDebugEnabled()) logger.debug("Extracting file %s to %
          s", file, destFile);
16     Files.copy(file, destFile, StandardCopyOption.REPLACE_EXISTING);
17     return FileVisitResult.CONTINUE;
18 ... //19 lines
19 }
```

approaches are less effective. Similar to vulnerability-detection approaches, existing API mapping approaches also overemphasize API bodies without being able to handle diverse implementations of functionality-equivalent APIs. Second, existing approaches are less generalizable because they are trained/fine-tuned on specific datasets, e.g., mapping first-party APIs in Java libraries to C# ones [29, 11, 6, 50] or Swift ones [47].

4 Approach

In this section, we propose APISS, the first approach to detect functionality-specific vulnerabilities via retrieving functionality-equivalent APIs. Unlike existing deep-learning and recurring-vulnerability detection approaches, APISS utilizes API doc strings and signatures to retrieve functionality-equivalent APIs and subsequently detect and exploit new functionality-specific vulnerabilities.

The workflow of APISS is illustrated in Figure 1. The input of APISS consists of two parts, an existing vulnerability (a.k.a., source vulnerability), including its affected API and its Proof-of-Concept (PoC), and a series of candidate APIs on which APISS intends to detect functionality-specific vulnerabilities if any exist. The output of APISS is a series of new vulnerabilities, each including an affected API and its corresponding PoC.

As shown in Figure 1, APISS contains two steps. First, APISS retrieves functionality-equivalent APIs among these candidate APIs by leveraging the embedding of each combination of API elements (doc string, signature, and body). Second, for each retrieved functionality-

[4] https://github.com/networknt/light-4j/pull/1768/commits/dff1dcb7f87622f86c953c9a7e639077d694a291

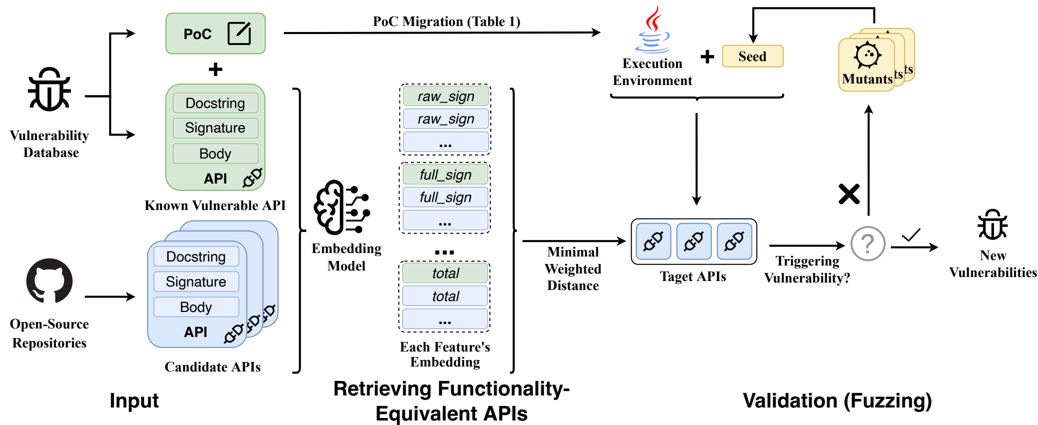

Figure 1 High-level framework of APISS.

equivalent API, APISS migrates the PoC of the source vulnerabilities. Additionally, for functionality-equivalent APIs corresponding to specific vulnerability types, APISS employs a fuzzing engine to conduct fuzzing based on the migrated PoC.

4.1 Retrieving Functionality-Equivalent APIs

Retrieving functionality-equivalent APIs is the basis of APISS as these APIs are more susceptible to vulnerabilities (similar to the given existing vulnerability) than other existing APIs.

Given that the source code of an API comprises three elements (its doc string, signature, and body), our key insight for retrieving functionality-equivalent APIs is to combine the embedding distances of these API elements. Specifically, we mainly focus on the embedding distances corresponding to API doc strings and signatures due to their similarities among functionality-equivalent APIs (as shown in Section 3.3). Considering the significant advancements achieved by Large Language Models (LLMs), we propose to leverage an LLM to embed various combinations of these three elements, termed as API features. To further investigate the contribution of each element, we also consider bodies when selecting features. In Section 5.4, we show that features including API bodies make only marginal contributions.

Specifically, we consider the following eight features:

- *raw_sign*: the raw signature, e.g., `public static void unzip (String zipFilename, String destDirname)` in Listing 2
- *full_sign*: the full signature (including the package and class names), e.g., `com.networknt.utility.NioUtils.unzip (String zipFilename, String destDirname)` in Listing 2
- *doc*: the doc string, e.g., Lines 1-7 in Listing 2
- *body*: the body, e.g., Lines 9-19 in Listing 2
- *raw_sb*: the combination of *raw_sign* and *body*
- *raw_ds*: the combination of *raw_sign* and *doc*
- *full_ds*: the combination of *full_sign* and *doc*
- *total*: the combination of *full_sign*, *doc*, and *body*

Pseudocode. Algorithm 1 presents the pseudocode of APISS' module of retrieving functionality-equivalent APIs. The algorithm takes as input a source API of an existing vulnerability ($srcAPI$), a set of candidate APIs ($candidateAPIs$), the weight of each feature ($\mathcal{W}_i$), and the desired number of functionality-equivalent APIs (N). This algorithm outputs N functionality-equivalent APIs associated with $srcAPI$.

Algorithm 1 Retrieving Functionality-Equivalent APIs.

```
input  : srcAPI, a source API (of an existing vulnerability)
input  : candidateAPIs, candidate APIs
input  : W_i, the weight of each feature
input  : N, target number of functionality-equivalent APIs
output : equivAPIs, srcAPI's functionality-equivalent APIs

   // Obtain one code line's minimal depth in AST
1  Function getEmbeddings (API) :
2      features ← [raw_sign, full_sign, doc, body, raw_sb, raw_ds, full_ds, total];
3      for i ∈ features do
           // Each feature's normalized embedding
4          E_i(API) ← LLM(API_i).last_hidden_states[−1];
5          Ê_i(API) = E_i(API) / ||E_i(API)||;
6      E(API) ← [Ê_1, ..., Ê_F];
7      return E(API)
   // Searching the top N closest APIs
8  E(src) ← getEmbeddings(srcAPI);
9  for dst ∈ candidateAPIs do
10     E(dst) ← getEmbeddings(dst);
11     D(src, dst) ← Σ_{i∈features} W_i * ||E_i(src) − E_i(dst)||_2;
12 equivAPIs ← argmin_dst[N]{D(src, dst)};
13 return equivAPIs;
```

In Lines 1–7, we compute each feature's embedding of a given API. In Line 4, we compute the raw embedding of each API feature, $\mathcal{E}_i(API)$. Then in Line 5, we normalize each embedding vector, denoted as $\hat{\mathcal{E}}_i(API)$. Our rationale for normalization is to ensure that the distances of each feature are within the same ranges.

In Lines 8–13, we compare the API embeddings of $srcAPI$ with those of $candidateAPIs$. In Lines 8–10, we generate the embeddings of both the $srcAPI$ and each candidate API using the `getEmbeddings` method. Then in Line 10, we define the distance of each pair of API embeddings as the weighted distance of each feature. Here, the distance of each feature is defined as the L2 norm [27, 12] of their embeddings' subtraction, $||\mathcal{E}_i(src) - \mathcal{E}_i(dst)||_2$.

Weight Optimization. The performance of Algorithm 1 depends on the selection of feature weights ($\mathcal{W}_i$). To determine the optimal set of values, we employ a global optimization technique. Specifically, we utilize the Basin-hopping algorithm [24], which has been widely applied in high-dimensional search problems [33, 48]. The optimization is performed on a dataset comprising Java and C# APIs (as shown in Table 6), and the resulting weights are subsequently used in the vulnerability detection process.

Acceleration by Vector Database. To enhance retrieval efficiency and mitigate the runtime costs of Algorithm 1, we utilize a vector database [20, 14]. Algorithm 1 requires calculating $|srcAPIs|$ * $|candidateAPIs|$ pairs of distances for all $|srcAPIs|$ as the distance between the src API and each dst API is computed in Lines 9-11. On the contrary, a vector database substantially reduces the runtime cost to $|srcAPIs| + |candidateAPIs|$ by storing

and retrieving the embeddings of $candidateAPIs$ while querying each $srcAPI$. During implementation, we choose Faiss[5] as our vector database and modify the selection criteria equivalently in Lines 11-12:

$$\textbf{Define: } \hat{\mathcal{E}}(API) = < Weight_1 * \hat{\mathcal{E}}_1, \ldots, Weight_F * \hat{\mathcal{E}}_F >$$
$$\underset{dst \in APIs}{argmin} \sum_{i \in Features} Weight_i * ||\hat{\mathcal{E}}_i(src) - \hat{\mathcal{E}}_i(dst)||_2 \quad (1)$$
$$\Leftrightarrow \underset{dst}{argmin} \, ||\hat{\mathcal{E}}(src) - \hat{\mathcal{E}}(dst)||_2$$

where $\mathcal{E}_i(API)$ is the i-th feature embedding of API, $\hat{\mathcal{E}}_i(API)$ is the normalized embedding, and $\hat{\mathcal{E}}(API)$ is the concatenated embeddings of all API's features. Thus, we store only $\hat{\mathcal{E}}(dst)$ of all candidate APIs in our vector database.

4.2 Migrating PoCs of Existing Vulnerabilities

In this step, we migrate the PoC for each retrieved functionality-equivalent API from that of its source API. Our goal is to reduce the substantial human labor of manually constructing PoCs. Specifically, the input for this step includes a PoC from a source API and one of its functionality-equivalent APIs, while the output is an executable PoC for this functionality-equivalent API.

Our Observation. Our solution leverages our observation that a functionality-specific vulnerability's PoC generally consists of two parts: input construction and API invocation. Input construction, i.e., creating specific objects or data structures, remains consistent across functionality-equivalent APIs. For example, in the PoCs of CVE-2023-27603 and CVE-2023-33484 (Listing 3), the construction of a zip file containing a malicious file is identical (Lines 12–30 in both PoCs). This zip file attempts to access a restricted path using the relative address "../../a/b/c/poc.txt". On the contrary, API invocation, i.e., the actual invocation of the targeted API, varies minimally and comprises a smaller code portion of the PoC. Lines 7–8 demonstrate that the "unzip" method is called in the former CVE and "NioUtils.unzip" is called in the latter, both aiming to unzip the malicious zip file. Notably, the parameters for these invocations, including the path of the zip file and the targeted extraction path, remain identical/similar across different APIs.

Migration Procedure. The preceding observation highlights the feasibility of adapting PoCs between functionality-equivalent APIs with minimal adjustment. Thus, we design a semi-automatic schema to migrate the PoC of existing vulnerabilities with two steps. First, for each type of vulnerability, we collect a representative affected API with its PoC, and manually create a heuristic-based template for PoC migration. The details are listed in Table 1. Second, for the API invocation part, we migrate the usage of each source API into the destination API based on a simple parameter-matching algorithm [45]. As for those cases where automatic migration fails, we manually migrate them. Specifically, we look up the documents in their corresponding repositories or directly consult with search engines.

[5] `https://faiss.ai/index.html`

Listing 3 The PoC of CVE-2023-27603 and CVE-2023-33484.

```
1 //API invocation
2 public static void main(String[] args) throws IOException {
3     zip();
4     String zipFilename = "./testData/unzip/poc.zip";
5     String destDirname = "./testData/unzip";
6     //The core lines of both PoCs
7     unzip(zipFilename, destDirname)(fs) // CVE-2023-27603
8     NioUtils.unzip(zipFilename, destDirname);  // CVE-2023-33484
9 }
10
11 // Input construction
12 public static void zip() {
13     ZipOutputStream zos = null;
14     try {
15         zos = new ZipOutputStream(new FileOutputStream( "./testData/
               unzip/poc.zip"));
16         String srcFile = "../../a/b/c/poc.txt";
17         String destFile = "./testData/unzip/poc.txt";
18         zos.putNextEntry(new ZipEntry(srcFile));
19         FileInputStream in = new FileInputStream(destFile);
20         int len;
21         byte[] buf = new byte[1024];
22         while ((len = in.read(buf)) != -1) {
23             zos.write(buf, 0, len);
24         }
25         zos.closeEntry();
26         in.close();
27     } catch (Exception e) {
28         throw new RuntimeException("zip␣error␣from␣ZipUtils", e);
29     } ...//9 Lines
30 }
```

API Fuzzing. We employ API fuzzing to further exploit functionality-specific vulnerabilities in functionality-equivalent APIs whose source APIs are associated with specific vulnerability types. Our rationale is that identical inputs may not trigger the same or similar vulnerabilities in different APIs if their functionalities diverge slightly. Thus, APISS employs a fuzzer to generate more inputs to exploit potential vulnerabilities. We focus our fuzzing efforts on APIs whose inputs are amenable to mutation, as some constructed inputs cannot be effectively fuzzed. The vulnerabilities whose affected APIs require fuzzing are detailed in the "Need Fuzzing" column of Table 1.

5 Evaluation

Our evaluation answers the following three research questions about APISS:

- **RQ1**: How effective is APISS in detecting real-world vulnerabilities when compared with baseline approaches?
- **RQ2**: How effective is APISS' API mapping module when compared with baseline approaches?
- **RQ3**: What are the manual/runtime costs of APISS?

Table 1 Heuristics of PoC Migration for Various Types of Vulnerabilities.

CWE Type	Characteristics of PoC	Fuzzing
CWE-121	Reuse the deep-nested JSON file in the "load" API	✓
CWE-22	Reuse the malicious zip file in the "unzip" API	✗
CWE-400/835	Reuse the test case with a vulnerable method sequence	✓
CWE-200	Reuse the malicious file in the "download" API	✗
CWE-502/611	Reuse the vulnerable JSON/XML files in the "parse" API	✓
CWE-94	Reuse the injected code in the execution API	✗
CWE-358	Reuse the initialization API of `collections` with various sizes	✓
CWE-434	Reuse the malicious file in the "upload" API	✗
CWE-275/79	Reuse the malicious XSS request in the "content_editing" API	✗
CWE-122/917	Reuse the malicious string expression in the "parse" API	✗

5.1 Background Information of Our Manual Efforts

Considering that the reliability of our detected vulnerabilities and the costs of our manual efforts highly depend on the expertise of our engineers, we explain the background of our manual efforts here. Evaluating APISS involves 10 software engineers, each with at least two years of Java development and one year of vulnerability-related tasks, e.g., vulnerability mining and fuzzing. Specifically, our engineers come from one of the major IT companies. This company has more than 20,000 engineers and more than 100 papers in top-tier conferences and journals on software engineering and security. The entire process of manually verifying existing vulnerabilities and their PoCs takes about 30 human-hours, and the entire process of detecting, validating, and reporting new vulnerabilities takes about 50 human-hours.

5.2 Dataset Collection

To evaluate APISS, we collect our dataset with two components: the affected APIs and PoCs of existing vulnerabilities, and a set of candidate APIs.

Vulnerability and PoC Collection. We collect source vulnerabilities from a Java vulnerability dataset used by VulFixMiner [51], MiDas [30], and CompVPD [10], including 1,359 vulnerabilities from 127 Java repositories. For each vulnerability in this dataset, we search for its related issue from the commit message in its patch, and the reference links in its vulnerability reports in NVD[6]. Then, for each issue with an executable PoC, we manually validate this PoC. For each type of functionality vulnerability, we choose the earliest PoC as the template one for migration (as shown in Table 2).

Candidate-API Collection. We extract 42,612 candidate APIs from 180 randomly sampled Java repositories on GitHub. Specifically, we adopt tree-sitter[7], a widely-used parser, to extract the elements of each API, including doc string, signatures, and bodies.

[6] https://nvd.nist.gov
[7] https://tree-sitter.github.io/tree-sitter/

Table 2 Each CWE Type's Source API.

CWE Type	CVE ID	Repository	API
CWE-121	CVE-2020-36518	jackson-databind	createContextual
CWE-22	CVE-2023-27603	apache_linkis	fileToUnzip
CWE-400/835	CVE-2018-18854	spray-json	JsObject
CWE-200	CVE-2021-21364	swagger-codegen	prepareDownloadFile
CWE-502/611	CVE-2018-21234	jodd	map2bean
CWE-94	CVE-2021-41269	cron-utils	parse
CWE-358	CVE-2018-1275	spring-framework	filterSubscriptions
CWE-434	CVE-2019-17352	jfinal	parse
CWE-275/79	CVE-2021-23899	json-sanitizer	sanitizeString
CWE-122/917	CVE-2022-24847	geoserver	getJNDIDataSource

Table 3 New Vulnerabilities Detected by APISS (TP and FP are True Positives and False Positives).

CWE Type	New CVE ID	Confirmed Without CVE ID	During Reporting	Reported by Others	TP	FP	Precision
CWE-121	24	8	59	7	98	53	0.65
CWE-22	2	7	2	0	11	31	0.26
CWE-400/835	14	0	15	0	29	8	0.78
CWE-200	2	0	0	0	2	37	0.05
CWE-502/611	1	2	0	3	6	24	0.20
CWE-94	3	0	0	2	5	33	0.13
CWE-358	0	0	11	0	11	17	0.39
CWE-434	3	0	0	0	3	15	0.17
CWE-275/79	10	0	0	2	12	12	0.50
CWE-122/917	1	1	0	0	2	33	0.06
Total	60	18	87	14	179	263	0.40

5.3 RQ1: How effective is APISS in detecting real-world vulnerabilities when compared with baseline approaches?

This research question aims to show the effectiveness of APISS in terms of detecting new vulnerabilities in real-world settings. Then, we take the vulnerabilities detected by APISS as ground truths to investigate the effectiveness of baselines (i.e., how many of them can be detected by our baselines).

Baselines. We compare APISS with three state-of-the-art deep-learning models: GGNN [23] (used by Devign [52], Funded [38], and ReVeal [8]), GCN [21] (used by IVDetect [22]), and RGCN [9] (used by Vu1SPG [49]). Although these models are originally trained on C/C++ vulnerability datasets, we retrain them on the preceding Java vulnerability dataset to ensure a fair comparison in detecting Java vulnerabilities. We also compare APISS with another state-of-the-art recurring-vulnerability detection approach, MVP [43]. We exclude AHPH [26] and ReurScan [36] as they are specialized for detecting specific recurring vulnerabilities (non-functionality ones).

Table 4 Vulnerabilities Detected by APISS with CVE IDs.

CVE ID	CWE Type	Repository	CVE ID	CWE Type	Repository
CVE-2023-33484	CWE-22	light-4j	CVE-2023-36067	CWE-121	groovy-json
CVE-2023-29638	CWE-79	WinterChenS/my-site	CVE-2023-36068	CWE-121	groovy-xml
CVE-2023-29641	CWE-79	pandao/editor	CVE-2023-36068	CWE-121	groovy-xml
CVE-2023-29643	CWE-79	perfree/PerfreeBlog	CVE-2023-35104	CWE-121	logansquare
CVE-2023-29636	CWE-79	ZHENFENG13/My-Blog	CVE-2023-35103	CWE-121	progsbase
CVE-2023-29639	CWE-79	ZHENFENG13/My-Blog	CVE-2023-35110	CWE-121	jjson
CVE-2023-29637	CWE-79	Qbian61/forum-java	CVE-2023-35116	CWE-121	jackson
CVE-2023-29635	CWE-434	Antabot/White-Jotter	CVE-2023-36073	CWE-94	Icefrog-core
CVE-2023-29652	CWE-434	RuoYi-Vue	CVE-2023-36074	CWE-94	bus-core
CVE-2023-33544	CWE-22	hawtio	CVE-2023-36071	CWE-121	com.grunka.json
CVE-2023-33546	CWE-121	janino-compiler/janino	CVE-2023-36075	CWE-611	ueboot-core
CVE-2023-33695	CWE-200	hutool	CVE-2023-41574	CWE-835	json-simple
CVE-2023-34670	CWE-200	nutzam/nutz	CVE-2023-42277	CWE-835	hutool-json
CVE-2023-29874	CWE-79	LovebuildJ/book-manager	CVE-2023-42276	CWE-835	hutool-json
CVE-2023-29875	CWE-79	LovebuildJ/book-manager	CVE-2023-42278	CWE-835	hutool-json
CVE-2023-29876	CWE-79	LovebuildJ/book-manager	CVE-2023-47382	CWE-400	aalto-xml
CVE-2023-29877	CWE-275	LovebuildJ/book-manager	CVE-2023-47381	CWE-400	aalto-xml
CVE-2023-29878	CWE-434	vran-dev/databasir	CVE-2023-50540	CWE-121	json-path
CVE-2023-34609	CWE-121	flexjson	CVE-2023-50546	CWE-121	json-path
CVE-2023-34610	CWE-121	json-io	CVE-2023-50570	CWE-835	ipaddress
CVE-2023-34612	CWE-121	ph-json	CVE-2023-50571	CWE-94	easy-rules-mvel
CVE-2023-34611	CWE-121	mjson	CVE-2023-50572	CWE-917	jline-groovy
CVE-2023-34615	CWE-121	JSONUtil	CVE-2023-51079	CWE-835	mvel
CVE-2023-34613	CWE-121	sojo	CVE-2023-51076	CWE-835	jxls
CVE-2023-34614	CWE-121	jsonij	CVE-2023-50635	CWE-835	hutool-all
CVE-2023-34616	CWE-121	progsbase	CVE-2023-50636	CWE-835	hutool-all
CVE-2023-34617	CWE-121	genson	CVE-2023-51074	CWE-121	json-path
CVE-2023-34620	CWE-121	hjson	CVE-2023-51075	CWE-835	hutool-core
CVE-2023-34623	CWE-121	jtidy	CVE-2023-51080	CWE-835	hutool-core
CVE-2023-34624	CWE-121	htmlcleaner	CVE-2023-51084	CWE-835	hyavijava

New Vulnerabilities Detected by APISS. At the time of writing, APISS has detected 179 new vulnerabilities. As shown in Table 3, 60 have received new Common Vulnerabilities & Exposures Identifiers (CVE IDs), 18 have been confirmed by developers without CVE IDs, 87 are currently under review, and 14 have been reported by others. Vulnerabilities with CVE IDs are listed in Table 4. Among the 60 vulnerabilities with new CVE IDs, 15 are classified as high-risk and 3 as critical based on the CVSS Base Score [35]. This result highlights APISS' effectiveness in detecting new vulnerabilities, bringing high value to security practice.

Difference Among Various Vulnerability Types. Table 3 shows that the majority of vulnerabilities detected by APISS belong to CWE-121 (Stack-based Buffer Overflow) and CWE-400/835 (Uncontrolled Resource Consumption/Infinite Loop). According to our manual investigation of the PoCs of these vulnerabilities, they are triggered by unexpected JSON inputs, especially those that are deeply nested. For example, Listing 5 shows the PoC of CVE-2023-34610, a new vulnerability (belonging to CWE-121) detected by APISS. Its input is constructed in Lines 3-22 and has the following format: "$\underbrace{[[\ldots[}_{9999}\,\texttt{0}\,\underbrace{]]\ldots]}_{9999}$".

APISS' False Positives. Table 3 also shows the false-positive numbers of each type of vulnerabilities. APISS incorrectly identifies 263 false-positive vulnerable APIs, achieving a high precision score of 0.40. It is mainly because we discard functionality-equivalent APIS for which APISS fails to generate its PoC.

Listing 4 An Example False-Positve Vulnerability.

```
1  import org.junit.Test;
2  import cn.hutool.core.net.*;
3  import java.util.List;
4  public class Ipv4UtilFuzzerList {
5      // oom
6      @Test
7      public void listFuzzerTest() {
8          try {
9              List<String> result = Ipv4Util.list("213.3.13.2/2/",
                   false);
10         } catch (Exception e) {
11         }
12     }
13 }
14
15 --------------------------------------------------
16 Error messages:
17 java.lang.OutOfMemoryError: Java heap space
18         at cn.hutool.core.net.Ipv4Util.list(Ipv4Util.java:141)
```

An Example of a False-Positive Vulnerability. Listing 4 presents an example of a false-positive vulnerability. The issue arises due to the use of a subnet mask of `/2`, which results in an excessively large capacity (i.e., 1,073,741,822) when initializing an ArrayList. The maintainer has acknowledged that a validation check could be added to Ipv4Util to handle such illogical inputs and prevent errors.

However, the maintainer does not classify this case as a vulnerability, stating that "A subnet mask is typically used to divide an IP address range into smaller networks, so a `/2` subnet is too large for any practical use, particularly for a public IP address such as `213.3.13.2`."

It is worth noting that false positives count for a small portion because most of them are crash-like vulnerabilities, which can be automatically verified. Such false positives rejected by repository maintainers account for a small fraction (i.e., less than 20%). Given the overall end-to-end effort (80 human-hours, as detailed in Section 5.1) and the effectiveness of our approach (identifying 179 new vulnerabilities and 60 new CVE IDs), the occurrence of such false positives remains within an acceptable range.

Comparison With Baselines. Table 5 illustrates the proportion of vulnerabilities detected by APISS and also detected by existing baselines. Overall, our baseline approaches can detect at most 58% of these vulnerabilities while MVP can detect 41% of them (corresponding to their recall values). Additionally, these baseline approaches encounter substantial false positives. MVP achieves a precision score of only 0.41 while the precision scores of existing deep-learning approaches (GCN, RGCN, and GGNN) are 0.02, 0.04, and 0.03, respectively. Such false positives will lead to a substantial manual costs when these identified vulnerabilities are validated. This result indicates that APISS is more effective in detecting functionality-specific vulnerabilities when compared with baselines. The main reason for their ineffectiveness is

[8] `https://nvd.nist.gov/vuln/detail/CVE-2023-34610`

Listing 5 The PoC of CVE-2023-34610.[8]

```java
public class PoC {
    //Input Construction
    public final static int TOO_DEEP_NESTING = 9999;
    public final static String TOO_DEEP_DOC = _nestedDoc(
        TOO_DEEP_NESTING, "[ ", "] ", "0");

    public static String _nestedDoc(int nesting, String open, String
        close, String content) {
        StringBuilder sb = new StringBuilder(nesting * (open.length()
             + close.length()));
        for (int i = 0; i < nesting; ++i) {
            sb.append(open);
            if ((i & 31) == 0) {
                sb.append("\n");
            }
        }
        sb.append("\n").append(content).append("\n");
        for (int i = 0; i < nesting; ++i) {
            sb.append(close);
            if ((i & 31) == 0) {
                sb.append("\n");
            }
        }
        return sb.toString();
    }
    //API invocation
    public static void main(String[] args) {
        String jsonString = TOO_DEEP_DOC;
        JsonReader.jsonToJava(jsonString);
    }
}
```

their overemphasis on API bodies disregarding the various implementations of functionality-equivalent APIs. On the contrary, APISS utilizes the similarities among API doc strings and signatures, thus successfully detecting these vulnerabilities.

5.4 RQ2: How effective is APISS' API mapping module when compared with baseline approaches?

This research question evaluates the effectiveness of APISS' API mapping module. In the evaluation, we choose two representative LLMs as the embedding model in APISS, ChatGPT (text-embedding-ada-002) [1] and CodeLLaMa (with 7B parameters) [34]. For each embedding model, we evaluate the effectiveness of each LLM with five feature combinations (configurations): `All Features` (using the optimal feature weights determined by Basin-hopping), `Signature+Doc String`, `Signature Only`, `Doc String Only`, and `Body Only`.

To quantitatively investigate the contribution of each embedding feature, we conduct Shapley Additive exPlanations (SHAPs) [3, 32], a game-theoretic technique to interpret the outputs of machine learning models. SHAP values help us understand how each feature contributes to APISS' effectiveness, thus providing evidence of focusing on doc strings and signatures instead of bodies.

Table 5 Proportions of Vulnerabilities Detected by Baselines.

CWE Type	MVP				GCN			
	TP	FP	Precision	Recall	TP	FP	Precision	Recall
CWE-121	52	50	0.51	0.53	18	874	0.02	0.18
CWE-22	5	4	0.56	0.45	4	141	0.03	0.36
CWE-400/835	10	9	0.53	0.34	5	116	0.04	0.17
CWE-200	0	8	0.00	0.00	0	1	0.00	0.00
CWE-502/611	0	4	0.00	0.00	3	144	0.02	0.50
CWE-94	2	0	1.00	0.40	1	44	0.02	0.20
CWE-358	4	20	0.17	0.36	1	75	0.01	0.09
CWE-434	0	0	0.00	0.00	0	0	0.00	0.00
CWE-275/79	0	9	0.00	0.00	0	0	0.00	0.00
CWE-122/917	0	3	0.00	0.00	0	0	0.00	0.00
Total	73	107	0.41	0.41	32	1,395	0.02	0.18

CWE Type	RGCN				GGNN			
	TP	FP	Precision	Recall	TP	FP	Precision	Recall
CWE-121	66	2,318	0.03	0.67	45	1,740	0.03	0.46
CWE-22	9	43	0.17	0.82	9	85	0.10	0.82
CWE-400/835	16	35	0.31	0.55	5	59	0.08	0.17
CWE-200	1	5	0.17	0.50	0	1	0.00	0.00
CWE-502/611	3	1	0.75	0.50	3	2	0.60	0.50
CWE-94	2	1	0.67	0.40	1	71	0.01	0.20
CWE-358	7	44	0.14	0.64	2	77	0.03	0.18
CWE-434	0	1	0.00	0.00	0	0	0.00	0.00
CWE-275/79	0	38	0.00	0.00	0	44	0.00	0.00
CWE-122/917	0	6	0.00	0.00	0	0	0.00	0.00
Total	104	2,492	0.04	0.58	65	2,079	0.03	0.36

Baselines. We select two state-of-the-art API mapping approaches as our baselines, Api2Api [29] and SAR [6]. Both approaches employ a seeding step that is critical to their performance. For fair comparison, we evaluate all three seeding strategies (used in their paper): Random Seeds, k-Fold Seeds ($k = 1, 2, 3, 4$), and Signature-Based Seeds. For each strategy, we choose the optimal seed number that maximizes accuracy, based on the highest reported results in their paper [6].

Dataset. We use the same dataset, Java2CSharp [18], as our baselines. This dataset maps Java First-Party APIs to functionality-equivalent C# First-Party APIs. It includes 1570 Java APIs, 2457 C# APIs, and 860 pairs of functionality-equivalent APIs.

Given that the API source code in Java2CSharp is not available and the SAR repository[9] does not provide complete details of API bodies and doc strings, we recollect all First-Party Java and C# APIs from *jdk8u40*[10] and *.NET 4.8. for Windows January 2020 Update*[11].

[9] `https://github.com/bdqnghi/SAR_API_mapping`
[10] `https://hg.openjdk.org/jdk8u/jdk8u40/jdk/file/c7bbaa04eaa8/src`
[11] `https://referencesource.microsoft.com/DotNet48ZDP2.zip`

Table 6 The Effectiveness of API Mapping.

Baselines	Configuration	Top-1	Top-5
Api2Api	Random Seeds	0.19	0.24
	1-Fold Seeds	0.24	0.35
	2-Fold Seeds	0.34	0.35
	3-Fold Seeds	0.37	0.51
	4-Fold Seeds	0.43	0.64
	Signature-based Seeds	0.35	0.41
SAR	Random Seeds	0.44	0.50
	1-Fold Seeds	0.36	0.39
	2-Fold Seeds	0.45	0.50
	3-Fold Seeds	0.54	0.66
	4-Fold Seeds	0.59	0.77
	Signature-based Seeds	0.54	0.75
APISS-CodeLLaMa	All Features	**0.69**	**0.78**
	Signature+Doc String	**0.69**	**0.78**
	Signature Only	0.65	0.70
	Doc String Only	0.02	0.03
	Body Only	0.19	0.31
APISS-ChatGPT	All Features	**0.81**	**0.84**
	Signature+Doc String	0.74	0.78
	Signature Only	**0.81**	**0.84**
	Doc String Only	0.34	0.44
	Body Only	0.40	0.58

For each configuration of baselines (Api2Api and SAR), we choose the optimal seed number that maximizes accuracy.

Metrics. To keep the same setting with our baselines, we use Top-k Accuracy ($k = 1, 5$) as our evaluation metric. Top-k Accuracy measures the accuracy of API mappings by assessing whether the functionality-equivalent C# API for a given Java API appears within the top k positions of the predicted API list.

Results of Top-k Accuracy. Table 6 shows the Top-k Accuracy results of APISS and baselines. APISS substantially improves the API mapping accuracy with an increase of 22% in Top-1 Accuracy ($0.81 - 0.59$) and an increase of 7% in Top-5 Accuracy ($0.84 - 0.77$). This improvement is attributed to APISS' use of LLMs, which effectively comprehend API functionalities.

We also observe that comparing the embeddings of only signatures achieves higher accuracy than our baselines, while comparing the embeddings of only doc strings or bodies achieves relatively lower accuracy. This result indicates that API signatures are the most critical feature for API mapping than doc strings or bodies.

Explanation of Each Feature's Contribution. The contribution of each feature in APISS' embedding models, ChatGPT and CodeLLaMa, are depicted in Figure 2 and Figure 3. Each feature's contribution percentage is noted within brackets, and the scatter points represent the feature's contribution on each input API.

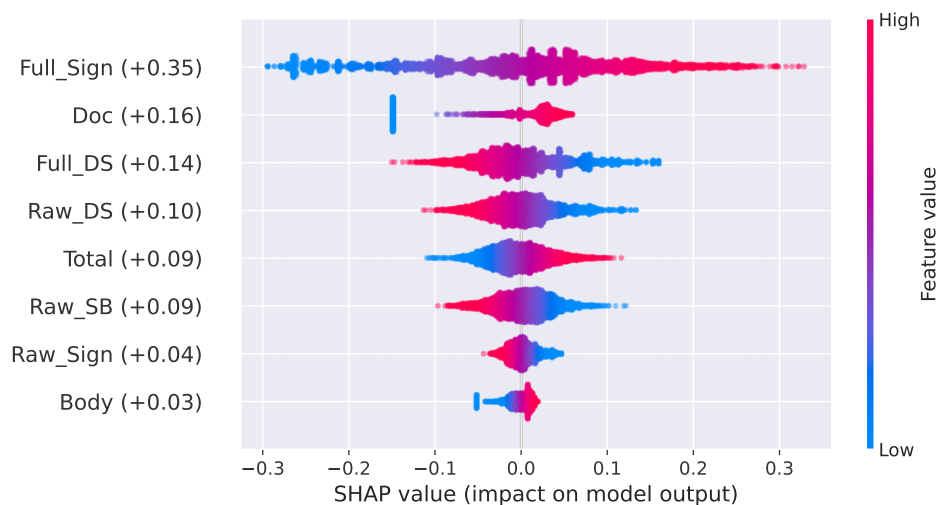

Figure 2 Each Feature's Contribution to API Mapping (Using ChatGPT's Embedding Model).

Both figures reveal that the three most important features are *full_sign*, *doc*, and *Full_DS*. These results highlight that both API doc strings and signatures (and their combination) mainly contribute to API mapping, while the API body has a negligible contribution (3% and 2%). Thus, the preceding results effectively support our proposal of utilizing API doc strings and signatures (instead of bodies) in API mapping.

Additionally, we notice two main differences from these figures. First, documentation strings show high value in smaller models in API mapping. For example, the most important feature for ChatGPT is *full_sign* (API signatures) while that of CodeLLaMa is *full_ds*, the combination of API doc strings and signatures. Second, features that include full signatures, which include package and class names (e.g., `com.networknt.utility.NioUtils`), are more effective than those with raw signatures. This result suggests that this contextual information enhances LLM's effectiveness in understanding API functionalities.

5.5 RQ3: What are the manual/runtime costs of APISS?

In this research question, we evaluate the manual and runtime costs of APISS. For manual costs, we detail the steps in the process where engineers detect new vulnerabilities, specifying their average time consumption. As for the runtime costs, we measure the end-to-end time costs of APISS on two main processes: (1) retrieving functionality-equivalent APIs, and (2) migrating PoCs and exploiting new vulnerabilities.

We perform all the evaluations on the system of Ubuntu 20.04. We use one Intel(R) Xeon(R) Gold 6248R@3.00GHz CPU, which contains 64 cores and 512GB memory. We use 1 Tesla A100 PCIe GPU with 40GB memory for API embedding.

Background Information of Our Manual Efforts. The end-to-end effectiveness of APISS highly depends on the expertise of our software engineers. Our evaluation, in which we have detected 179 new vulnerabilities, includes five software engineers. Each engineer has at least two years of experience in Java development and one year in fuzzing. The entire process of their manual costs takes approximately 50 man-days.

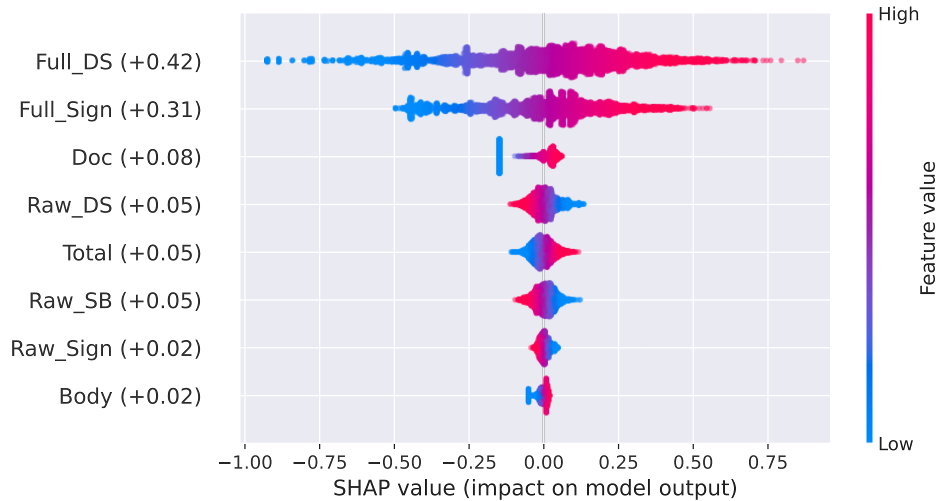

Figure 3 Each Feature's Contribution to API Mapping (Using CodeLLaMa for Embedding).

The Manual Costs of APISS. Figure 4 depicts the detailed workflow of how our engineers are involved in detecting new functionality-specific vulnerabilities. In this flowchart, steps requiring human intervention are highlighted in yellow. Adjacent to each step, we provide a breakdown of the average time costs.

The process of retrieving functionality-equivalent APIs, including extracting and validating the PoCs of existing vulnerabilities, requires approximately four hours per PoC. Our engineers read the vulnerability issue (if exists) and its URL links, and test the PoC within the affected software repository (on its vulnerable version). As shown in Section 5.3, a single validated PoC leads to the detection of an average of 17.9 new vulnerabilities, so the manual costs of four hours are affordable. Additionally, existing vulnerability databases, e.g., Snyk[12] and VeraCode[13], have already collected a large number of PoCs although they do not open-source these PoCs to avoid potential risks.

The procedure of migrating PoCs and exploiting new vulnerabilities involves three steps requiring human intervention. First, our engineers identify the usage for each functionality-equivalent API that cannot be directly invoked by our parameter-matching algorithm [45]. This task applies to 20% of APIs, with each requiring 3-5 minutes to address. Second, for each API triggered by a potential vulnerability, our engineers verify whether the API inputs are legal. This step is necessary for 19.6% of APIs, also taking 3-5 minutes per API. Third, once a vulnerability is confirmed, reporting each one takes approximately 15 minutes. We exclude the manual costs of this step as the vulnerability is already detected and exploited. Therefore, the manual costs of migrating a PoC to a new vulnerability is within 10 minutes.

The Runtime Costs of APISS. The runtime costs of APISS are shown in Table 7. For API mapping, APISS demonstrates high efficiency, with each step of embedding and retrieving taking less than one second. In the fuzzing process, we limit the maximum duration to two hours, determined based on the similarities between migrated inputs and potential vulnerability-triggering inputs. Consequently, the end-to-end fuzzing costs a total of 296 hours and successfully detects 144 vulnerabilities. These results confirm that the runtime costs of APISS are practical for real-world applications.

[12] https://snyk.io
[13] https://www.veracode.com

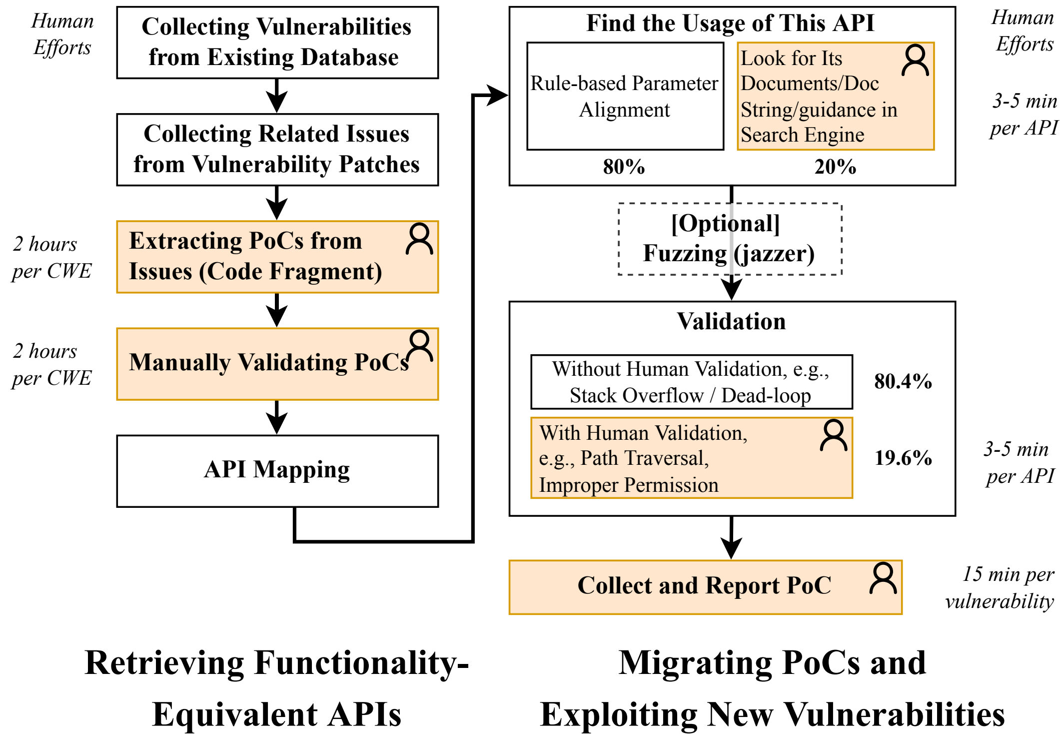

Figure 4 The Details of Human-Involved Procedure. White Boxes are Automatic Steps While Orange Boxes are Human-Involved Steps.

6 Discussion

Detecting Functionality-Specific Vulnerabilities of Other Programming Languages. We primarily focus on Java vulnerabilities due to the widespread usage of Java third-party libraries in software development [40]. APISS is designed with a high degree of generalization ability and is not restricted to Java vulnerabilities. For instance, APISS detects a new Java vulnerability (Listing 2) based on a Scala one (Listing 1). In Section 5.4, we also show that APISS can effectively retrieve functionality-equivalent APIs among various programming languages, e.g., Java and C#, without extra fine-tuning or training. Thus, APISS can be easily adapted to detect other programming languages' functionality-specific vulnerabilities.

Detecting Non-Functionality-Specific Vulnerabilities. Non-functionality-specific vulnerabilities also count for a large portion of vulnerabilities in third-party libraries. For example, among the Top 5 vulnerability types in 2023[14], CWE-787 (Out-of-bounds Write) and CWE-416 (Use-After-Write) are two representative non-functionality-specific ones because they do not occur in specific functionalities and remain widespread, especially in C/C++ repositories. Thus, the likelihood of being affected by these vulnerabilities is quite similar between functionality-equivalent APIs and the rest APIs. Now that APISS mainly focuses on retrieving candidate APIs based on their functionalities, we consider detecting such vulnerabilities as future work as they belong to different scopes of approaches.

[14] https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html

Table 7 Hyper-Parameters and Runtime Costs of APISS.

API Mapping	CodeLLaMa	ChatGPT
Embedding Length	4,096 (default)	1,536 (default)
Embedding	465ms (per API)	554 ms (per API)
Initialize DB	0.4ms (per API)	0.1 ms (per API)
Retrieving	391ms (per API)	423 ms (per API)
Fuzzing Runtime	2 hours (per API)	296 hours (Total)

7 Threats to Validity

A major threat to external validity comes from the training dataset for evaluating existing deep-learning approaches. Existing deep-learning approaches [52, 8] are trained on two datasets including only C/C++ repositories while the vulnerabilities detected by APISS are mainly Java ones. To mitigate this threat, we choose a widely-used Java vulnerability dataset [51, 30, 10], which is general enough by including 1,359 Java vulnerabilities from 127 Java repositories. Thus, our reproduced models of existing deep-learning approaches reflect their effectiveness in detecting Java vulnerabilities. Additionally, the vulnerabilities in this dataset are before 2022 while those detected by APISS are later in 2023, so the potential risks of look-ahead bias [2] also do not exist.

The threats to internal validity are instrumentation effects that can bias our results. To reduce these threats, we manually inspect the intermediate results such as the functionality-equivalent APIs retrieved by APISS for dozens of sampled vulnerability patches, such as those in Listing 1 and Listing 2.

8 Related Work

8.1 Vulnerability Detection Approaches

Recurring-Vulnerability Detection Approaches. These approaches [43, 26, 36] are designed to detect recurring vulnerabilities, which come from reused code or shared code logic. For example, MVP [43] utilizes hash values based on the code property graphs of known vulnerable APIs to match similar APIs in open-source repositories.

However, these approaches rely heavily on matching the preceding implementation-based features. These approaches may produce high false negatives in detecting functionality-specific vulnerabilities as they rely heavily on matching the preceding implementation-based features while the affected APIs have diverse implementations. On the contrary, APISS considers functionality equivalence, a less strict constraint than implementation equivalence, to detect more vulnerabilities.

Deep-Learning Approaches. These approaches [41, 52, 39, 8, 38, 37, 42] determine whether a given method contains a potential vulnerability by learning code features of vulnerable APIs/methods of existing vulnerabilities. Devign [52] is a representative deep-learning approach. It uses Code Property Graphs (e.g., control/data-flow graphs) to represent the given method and designs a specialized Graph Neural Network (GNN) to determine whether this method is vulnerable.

These approaches can effectively detect vulnerabilities caused by incorrect implementation, such as use-after-free, because the source code of their affected APIs has specific patterns, e.g., a pointer operation after a `free` API invocation. However, deep-learning approaches are ineffective in detecting functionality-specific vulnerabilities because their affected APIs have diverse implementations without a common pattern, and they also face limitations in generating PoCs for detected vulnerabilities. On the contrary, APISS specifies the targeted vulnerabilities as functionality-specific ones, whose PoCs can be migrated from existing vulnerabilities.

8.2 API Mapping Approaches

API Mapping maps a library API to another functionality-equivalent one. Existing work [29, 11, 6, 50, 47] mainly focuses on translating an API of a specific programming language to another one. For example, SAR [6] translates first-party Java APIs to C# ones. Specifically, it learns the vector representation of both Java and C# APIs and trains a mapping model by adversarial learning. However, there are two main limitations when adopting existing API mapping approaches to retrieve functionality-equivalent APIs. First, for each pair of programming languages, these approaches need to train a specific model to translate an API between them, resulting in the scalability problem. Second, similar to existing vulnerability detection approaches, they also overemphasize the importance of API bodies. As shown in Section 5.4, API bodies hardly contribute to the effectiveness of API mapping. Thus, we design our API mapping module in APISS to address the preceding limitations.

8.3 API Fuzzing

API fuzzing [16, 4, 15, 44, 13, 19, 28, 5], as a special category of fuzzing approaches [53, 5, 25], is widely used to exploit API vulnerabilities. RULF [19] is a representative approach that fuzzes API vulnerabilities in Rust standard libraries. These approaches mainly use template-based or synthesized API sequences and randomly generated API inputs. Additionally, these approaches need to be executed for a long period to cover the input space as possible, thus resulting in low efficiency/effectiveness. On the contrary, APISS migrates the PoCs and inputs of existing vulnerabilities, so it can detect and trigger new functionality-specific vulnerabilities with higher probability within a short period.

9 Conclusion

In this paper, we have presented the first approach to utilize API doc strings and signatures instead of focusing on API bodies to retrieve functionality-equivalent APIs and then detect functionality-specific vulnerabilities. We have conducted a comprehensive evaluation to demonstrate APISS' effectiveness in retrieving functionality-equivalent APIs, achieving a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55. We have designed a semi-automatic schema to reduce the manual costs of migrating the PoC of existing vulnerabilities. We have demonstrated APISS' efficiency, costing less than 10 minutes of human labor per vulnerability, and the end-to-end runtime overhead of analyzing one candidate API is less than 2 hours. We have demonstrated APISS' high value to security practice by detecting 179 new vulnerabilities and receiving 60 new CVE IDs.

References

1 ChatGPT, 2023. URL: https://openai.com/chatgpt.

2 Maya Almaraz, Michelle Y Wong, and Wendy H Yang. Looking back to look ahead: a vision for soil denitrification research. *Ecology*, 101(1):e02917, 2020.

3 Liat Antwarg, Ronnie Mindlin Miller, Bracha Shapira, and Lior Rokach. Explaining anomalies detected by autoencoders using shapley additive explanations. *Expert Systems with Applications*, 186:115736, 2021. doi:10.1016/J.ESWA.2021.115736.

4 Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 748–758, 2019. doi:10.1109/ICSE.2019.00083.

5 Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017. doi:10.1145/3133956.3134020.

6 Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 796–806, 2019.

7 Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. Coca: improving and explaining graph neural network-based vulnerability detection systems. *arXiv preprint arXiv:2401.14886*, 2024. doi:10.48550/arXiv.2401.14886.

8 Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2022. doi:10.1109/TSE.2021.3087402.

9 Junjie Chen, Hongxu Hou, Jing Gao, Yatu Ji, and Tiangang Bai. RGCN: recurrent graph convolutional networks for target-dependent sentiment analysis. In *Proceedings of International Conference on Knowledge Science, Engineering and Management*, pages 667–675, 2019. doi:10.1007/978-3-030-29551-6_59.

10 Tianyu Chen, Lin Li, Taotao Qian, Zeyu Wang, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. Identifying vulnerability patches by comprehending code commits with comprehensive change contexts. *arXiv preprint arXiv:2310.02530*, 2023. doi:10.48550/arXiv.2310.02530.

11 Lei Cui, Jiancong Cui, Yuede Ji, Zhiyu Hao, Lun Li, and Zhenquan Ding. API2Vec: learning representations of API sequences for malware detection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–273, 2023. doi:10.1145/3597926.3598054.

12 HWJ Debeye and P Van Riel. Lp-norm deconvolution. *Geophysical Prospecting*, 38(4):381–403, 1990.

13 Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational API inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 44–56, 2022. doi:10.1145/3540250.3549085.

14 Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: a survey. *arXiv preprint arXiv:2312.10997*, 2023.

15 Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020. doi:10.1145/3368089.3409719.

16 Harrison Green and Thanassis Avgerinos. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1070–1081, 2022. doi:10.1145/3510003.3510228.

17 Hazim Hanif and Sergio Maffeis. Vulberta: simplified source code pre-training for vulnerability detection. In *Proceedings of 2022 International Joint Conference on Neural Networks*, pages 1–8, 2022. `doi:10.1109/IJCNN55064.2022.9892280`.

18 Java2CSharp. https://github.com/codejuicer/java2csharp, 2017.

19 Jianfeng Jiang, Hui Xu, and Yangfan Zhou. RULF: Rust library fuzzing via API dependency graph traversal. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 581–592, 2021. `doi:10.1109/ASE51524.2021.9678813`.

20 Zhi Jing, Yongye Su, Yikun Han, Bo Yuan, Chunjiang Liu, Haiyun Xu, and Kehai Chen. When large language models meet vector databases: a survey. *arXiv preprint arXiv:2402.01763*, 2024.

21 Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. `arXiv:1609.02907`.

22 Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021. `doi:10.1145/3468264.3468597`.

23 Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

24 Zhenqin Li and Harold A Scheraga. Monte carlo-minimization approach to the multiple-minima problem in protein folding. *Proceedings of the National Academy of Sciences*, 84(19):6611–6615, 1987.

25 Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018. `doi:10.1109/TR.2018.2834476`.

26 Miaoqian Lin, Kai Chen, and Yang Xiao. Detecting API post-handling bugs using code and description in patches. In *Proceedings of the 32nd USENIX Security Symposium*, pages 3709–3726, 2023. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/lin`.

27 Xiong Luo, Xiaohui Chang, and Xiaojuan Ban. Regression and classification using extreme learning machine based on l1-norm and l2-norm. *Neurocomputing*, 174:179–186, 2016. `doi:10.1016/J.NEUCOM.2015.03.112`.

28 Riyadh Mahmood, Jay Pennington, Danny Tsang, Tan Tran, and Andrea Bogle. A framework for automated API fuzzing at enterprise scale. In *Proceedings of 2022 IEEE Conference on Software Testing, Verification and Validation*, pages 377–388, 2022. `doi:10.1109/ICST53961.2022.00018`.

29 Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering*, pages 438–449, 2017. `doi:10.1109/ICSE.2017.47`.

30 Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E. Hassan, Xuan-Bach D. Le, and David Lo. Multi-granularity detector for vulnerability fixes. *IEEE Transactions on Software Engineering*, 49(8):4035–4057, 2023. `doi:10.1109/TSE.2023.3281275`.

31 Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 178–182, 2022. `doi:10.1145/3510454.3516865`.

32 Yasunobu Nohara, Koutarou Matsumoto, Hidehisa Soejima, and Naoki Nakashima. Explanation of machine learning models using shapley additive explanation and application for real data in hospital. *Computer Methods and Programs in Biomedicine*, 214:106584, 2022. `doi:10.1016/J.CMPB.2021.106584`.

33 Gustavo G Rondina and Juarez LF Da Silva. Revised basin-hopping monte carlo algorithm for structure optimization of clusters and nanoparticles. *Journal of Chemical Information and Modeling*, 53(9):2282–2298, 2013. doi:10.1021/CI400224Z.

34 Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. CodeLlama: open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

35 Karen Scarfone and Peter Mell. An analysis of CVSS version 2 vulnerability scoring. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 516–525, 2009. doi:10.1109/ESEM.2009.5314220.

36 Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. RecurScan: detecting recurring vulnerabilities in PHP web applications. In *Proceedings of the ACM Web Conference 2024*, pages 1746–1755, 2024. doi:10.1145/3589334.3645530.

37 Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3623345.

38 Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020. doi:10.1109/TIFS.2020.3044773.

39 Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. DeepVD: toward class-separation features for neural network vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering*, pages 2249–2261, 2023. doi:10.1109/ICSE48619.2023.00189.

40 Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *Proceedings of 2020 IEEE International Conference on Software Maintenance and Evolution*, pages 35–45, 2020. doi:10.1109/ICSME46990.2020.00014.

41 Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M Zhang, and Qing Liao. Vulnerability detection with graph simplification and enhanced graph representation learning. *arXiv preprint arXiv:2302.04675*, 2023. doi:10.48550/arXiv.2302.04675.

42 Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 345–357, 2023. doi:10.1109/ASE56229.2023.00144.

43 Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures. In *Proceedings of the 29th USENIX Security Symposium*, pages 1165–1182, 2020.

44 Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. Docter: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–188, 2022. doi:10.1145/3533767.3534220.

45 Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: inference and application of API migration edits. In *Proceedings of 2019 IEEE/ACM 27th International Conference on Program Comprehension*, pages 335–346, 2019. doi:10.1109/ICPC.2019.00052.

46 Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. Enhancing LLM-based test generation for hard-to-cover branches via program analysis. *arXiv preprint arXiv:2404.04966*, 2024. doi:10.48550/arXiv.2404.04966.

47 Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. Deep-diving into documentation to develop improved Java-to-Swift API mapping. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 106–116, 2020. doi:10.1145/3387904.3389282.

48 Yafan Zhao, Xin Chen, and Jun Li. Tgmin: A global-minimum structure search program based on a constrained basin-hopping algorithm. *Nano Research*, 10:3407–3420, 2017.

49 Weining Zheng, Yuan Jiang, and Xiaohong Su. Vu1SPG: vulnerability detection based on slice property graph representation learning. In *Proceedings of 2021 IEEE 32nd International Symposium on Software Reliability Engineering*, pages 457–467, 2021. `doi:10.1109/ISSRE52982.2021.00054`.

50 Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204, 2010. `doi:10.1145/1806799.1806831`.

51 Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. Finding a needle in a haystack: automated mining of silent vulnerability fixes. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 705–716, 2021. `doi:10.1109/ASE51524.2021.9678720`.

52 Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

53 Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys*, 54(11s):1–36, 2022. `doi:10.1145/3512345`.