Securing Self-Managed Third-Party Libraries

Xin Zhou*, Jinwei Xu*, He Zhang*, Yanjing Yang*, Lanxin Yang*, Bohan Liu*, Hongshan Tang†

*State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University, China
†JD.com, Inc., China
Corresponding author: Xin Zhou, zhouxin@nju.edu.cn

***Abstract*—Modern software development reuses third-party libraries to cut costs but may introduce vulnerabilities. A critical practice is to verify the security of third-party libraries against public vulnerability reports. Many automated methods have been proposed to identify vulnerable libraries from vulnerability reports. Existing methods are designed for the generic identification of vulnerable libraries, considering the security of all software libraries. Generic identification is inherently challenging, resulting in limited accuracy. However, organizations only consider the security of libraries they trust and use, by self-managing a library whitelist. Therefore, we propose LibGuard, a framework to adapt existing methods to help organizations secure the libraries they use. LibGuard supplies a library whitelist for existing methods and filters the results according to a threshold, facilitating the discovery of risks overlooked by organizations while controlling false alarms. LibGuard is implemented in two ways. The first attaches the whitelist after existing methods. The second integrates the whitelist into existing methods. We evaluated LibGuard using 5,107 vulnerability reports and the library whitelist built from 79 Google projects and 29 Huawei projects. The results show that the two implementations of LibGuard increase the average F1 score by 10.25% and 11.77%, respectively. Moreover, LibGuard performs stably during the extension of whitelists. To our knowledge, this paper is the first study dedicated to securing self-managed third-party libraries, offering insights into adapting generic software security management to self-managed contexts.**

I. Introduction

In modern software development, third-party libraries are frequently reused to improve efficiency and reduce costs [1, 2]. While leveraging third-party libraries offers significant advantages, it introduces potential risks [3, 4, 5], particularly concerning vulnerabilities in dependencies (*i.e.*, the third-party libraries on which the software depends) [6]. Thus, developers must scrutinize the security of the dependencies they incorporate [7].

A prevalent security practice is to check dependencies against known vulnerabilities reported in public databases such as the National Vulnerability Database (NVD) [8]. However, NVD reports may not explicitly indicate vulnerable libraries [7]. In addition, the data provided in NVD reports might be inaccurate or lacking details [9, 10, 11]. Therefore, manually identifying vulnerable libraries from vulnerability reports is labor intensive and prone to errors [12]. To address this challenge, many methods have been developed to help identify vulnerable libraries from vulnerability reports [12, 13].

The past few years have seen many studies [7, 12, 13, 14, 15] that proposed methods to automatically identify vulnerable libraries from public vulnerability reports. According to the identification mechanism, these methods can be divided into three categories: (1) *XML-based* (eXtreme Multi-label Learning-based), (2) *DL-based* (Deep Learning-based), and (3) *LLM-based* (Large Language Model-based). One representative method of *XML-based* is Chronos [12], which utilizes an XML model to map vulnerability reports to libraries in the vulnerable library pool it predefined. One representative method of *DL-based* is VulLibMiner [14], which utilizes a deep learning model to identify vulnerable libraries out of all Maven libraries based on vulnerability reports. One representative method of *LLM-based* is VulLibGen [13], which utilizes an LLM to generate the names of vulnerable libraries based on vulnerability reports and outputs of VulLibMiner.

After interviewing 19 practitioners from 5 organizations across diverse sectors, including IT, finance, and cloud service providers, we find that existing methods are not compatible with the usage requirements of organizations. Existing methods identify vulnerable libraries out of all libraries, *e.g.*, all libraries in the Maven repository [13, 14], while organizations are primarily concerned with the security of libraries they currently use, rather than all potentially vulnerable libraries [4, 11, 16, 17, 18]. Therefore, for organizations, the generic design of existing methods unnecessarily increases the task difficulty and reduces the identification accuracy. For instance, VulLibGen [13] identifies *commons-io.wso2:commons-io* as the vulnerable library of CVE-2024-47554, but in fact, the vulnerable library is *commons-io:commons-io*. If an organization depends on *commons-io:commons-io*, it will not receive alerts and will continue to be under security threats. Existing methods consider a wide range of libraries, including many with similar names, which reduces the accuracy of identification. To provide more details of our interview, we have released the detailed interview protocol, the full list of guiding questions, and an anonymized summary of practitioner feedback in the supplementary material.

In addition to the problem of low accuracy, the practitioners note that although automated methods encompass a large library pool, some libraries their organizations depend on are not included, implying that existing methods can not cover the security of all libraries they use. For instance, VulLibMiner [14] encompasses a library pool of 435,642 Maven libraries. However, some Maven libraries depended on by the *google/guava* project are not included in the library pool, *e.g.*, *org.codehaus.plexus:plexus-io* and *org.mvnsearch: toolchains-*

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00307

maven-plugin.

To summarize, existing methods suffer from low accuracy and inadequate coverage of organizations' dependencies, leaving organizations unaware of potential security threats. Actually, organizations ensure the security of their dependencies by managing a library whitelist, *i.e.*, the third-party libraries they trust and allow for use. In this paper, the term 'whitelist' is used in this sense, which is equivalent to the more inclusive term 'allowlist' [19]. Self-managing a library whitelist requires continuous monitoring and interacting with external vulnerability information. The practitioners interviewed highlight that this measure is crucial in light of incidents such as the Apache Struts 2 vulnerability [20], underscoring the need for vigilance even with libraries previously trusted. Moreover, practitioners indicate that as organizations evolve, they continuously expand their library whitelists to ensure that security is maintained along with business growth. Therefore, an ideal method for organizations should focus on the security of libraries in the whitelist, issuing alerts timely when organizations are affected by vulnerabilities, minimizing false alarms when they are not, and adapting to the continuous expansion of the whitelist.

In this paper, we propose **LibGuard**, a framework for securing self-managed third-party libraries. LibGuard consists of three components: (1) identification of vulnerable libraries, (2) supply of library whitelists, and (3) filtering based on thresholds. The first component identifies vulnerable libraries from vulnerability reports. The second component limits the identification scope to the whitelist, reducing the task difficulty and improving the accuracy. The third component filters the previous output based on a threshold to identify vulnerable libraries overlooked by organizations and control false alarms at the same time. LibGuard is implemented in two ways. The first attaches the whitelist after vulnerable library identification methods. Then it measures the similarity between the identified library and each library in the whitelist. An alert is triggered based on a predefined similarity threshold. The second integrates the whitelist into identification methods to narrow the identification scope. Then it calculates the prediction score for each library in the whitelist and triggers an alert based on a predefined score threshold.

We conducted experiments using 5,107 vulnerability reports and 108 projects from two organizations (79 from Google and 29 from Huawei) to compare LibGuard against three baselines [12, 13, 14]. The results show that existing methods are limited in securing self-managed libraries for organizations. By comparison, LibGuard with whitelist attachment improves the F1 score from 56.77% to 67.01% when built on VulLibGen, with a false alarm ratio of 12.26%. Moreover, LibGuard with whitelist integration achieves the F1 score of 68.54% when built on VulLibGen, with a false alarm ratio of 16.69%. When the library whitelist is expanded, LibGuard maintains stable performance.

The contributions of this paper are highlighted as follows.

- We identify a research gap in identifying vulnerable libraries aimed at securing self-managed library whitelists for organizations.
- We propose LibGuard, a framework for securing self-managed third-party libraries, and conduct extensive experiments to evaluate it.
- We disclose a dataset consisting of the whitelists of 108 real-world projects from two organizations for future research.

II. Background and Related Work

A. Third-Party Library Security Management

Third-party libraries are prevalent in software development, allowing developers to reuse desired functionalities without reinventing the wheel [21, 22]. However, this code-sharing culture leads to one critical problem: the propagation of vulnerabilities in third-party libraries [23, 24, 25, 26]. For example, the Log4Shell vulnerability in the widely used Apache Log4j library affected millions of organizations such as Amazon, Cloudflare, and Tencent, causing catastrophic consequences [27, 28]. Therefore, the security of third-party libraries needs to be properly managed.

Software Composition Analysis (SCA) tools, such as Veracode [29] and Snyk [30], have been proposed to help manage the security of third-party libraries. SCA tools rely on a vulnerability database to associate disclosed vulnerability with vulnerable libraries [31]. The database is maintained by security experts assisted by automated tools. Security experts identify vulnerable libraries from public vulnerability reports based on their domain knowledge and the results of automated tools [12, 32]. However, according to Haryono et al. [7], 53.3% of vulnerability reports do not explicitly indicate affected libraries. More than half of the vulnerable libraries disclosed in reports are incomplete or inaccurate [33, 34]. These problems challenge the accuracy of identifying vulnerable libraries from vulnerability reports, whether through manual or automated methods, limiting the effectiveness of SCA tools.

Organizations place a high priority on the security of third-party libraries. They self-manage the security of their dependencies based on limited library whitelists [35], as they are primarily concerned with the security of the libraries they depend on, rather than the security of all third-party libraries [4, 16]. As the limited libraries it depends on, an organization is only affected by a small number of disclosed vulnerabilities [36, 37]. Considering the cost of manual verification is an important concern for organizations [38, 39, 40], a tool to help organizations manage their library whitelists should timely issue alarms when vulnerable libraries are in the whitelist while remaining silent (*i.e.*, no false alarms) when vulnerable libraries are not in the whitelist.

B. Vulnerable Library Identification

There have been extensive studies on the automated identification of vulnerable libraries. According to their identification mechanisms they can be classified into three categories: (1) *XML*-based, *e.g.*, FastXML [15], LightXML [7], Chronos [12], (2) *DL*-based, *e.g.*, VulLibMiner [14], and (3) *LLM*-based, *e.g.*, VulLibGen [13].

Chen et al. [15] applied FastXML to address the task of identifying vulnerable libraries from NVD reports. They are the first to conceptualize this task as an XML problem and train FastXML to automate its solution. FastXML achieves an average F1 score of 0.51 in 7,696 NVD reports. Haryono et al. [7] evaluated a variety of popular XML models in identifying vulnerable libraries from NVD reports. The results show that LightXML performs better than other XML models such as FastXML [15], DiSMEC [41], and ExtremeText [42]. LightXML achieves an average F1 score of 0.70. Lyu et al. [12] proposed Chronos which is based on ZestXML to identify vulnerable libraries from NVD reports in chronological order. Chronos utilizes ZestXML to identify vulnerable libraries that are unseen during training. Chronos achieves an average F1 score of 0.75 on all samples and an average F1 score of 0.70 on samples with unseen libraries.

Chen et al. [14] proposed VulLibMiner, which combines a TF-IDF matcher and a BERT-FNN model to accurately identify vulnerable libraries from vulnerability reports and textual descriptions of libraries. The TF-IDF matcher first filters a small set of candidate libraries, which are then further evaluated using the trained BERT-FNN model. Compared with previous work providing names of libraries, VulLibMiner supplements descriptions of libraries, alleviating the problem of confusing libraries with complex and similar names. In evaluations on the VeraJava and VulLib datasets [43], VulLibMiner achieved an average F1 score of 0.56.

Chen et al. [13] proposed VulLibGen, which utilizes large language models (LLMs) with prior knowledge of extensive libraries to identify vulnerable libraries from vulnerability reports. They employ input augmentation techniques by feeding the top-k of vulnerable libraries (recommended libraries) identified by VulLibMiner into the LLM to assist in identifying vulnerable libraries that are not encountered during training. Post-processing techniques are used to address the hallucination problem of LLMs, *i.e.*, generating seemingly plausible but non-existent library names). VulLibGen demonstrated outstanding performance on the VulLib dataset [43], with an average F1 score of 0.63.

In summary, existing methods are designed for generic identification of vulnerable libraries and manage the software security for public, ignoring the private contexts of securing self-managed libraries. Therefore, they generate inaccurate alerts when managing a library whitelist for an organization.

III. Motivating Example

This section describes two examples to motivate this study.

As shown in Figure 1, the first example illustrates the challenge of overlooking vulnerable libraries by misidentifying those with similar names (**Challenge 1**). The automated model (VulLibMiner) identifies *org.apache.hive:hive-metastore* as the vulnerable library of CVE-2022-41137, while the actual vulnerable library of this vulnerability is *org.apache.hive:hive-exec*. Although *org.apache.hive:hive-exec* exhibits a relatively high prediction score, *org.apache.hive:hive-metastore* receives a higher score than it. Therefore, the model outputs an

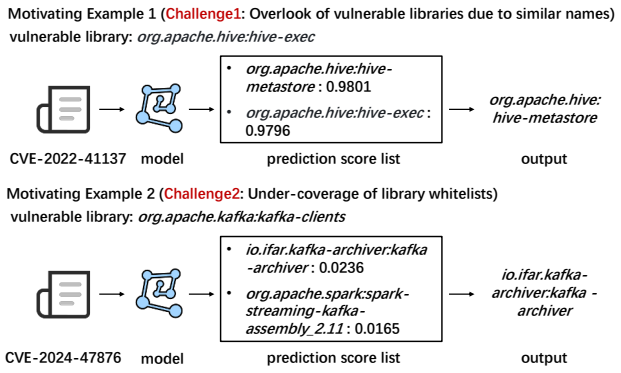

Fig. 1. The motivating examples

incorrect result, overlooking the actual vulnerable library. The model identifies vulnerable libraries out of all the Maven libraries, which is difficult as there are many Maven libraries with similar names. In this example, the model tries to identify *keycloak-saml-core* from *keycloak-saml-api*, *keycloak-saml-core-public*, and *keycloak-core*, etc. The vulnerability report of CVE-2024-4629 does not mention the vulnerable library explicitly, therefore the model fails to identify the vulnerable library correctly. If an organization depends on *keycloak-saml-core*, it will not receive an alert and will remain under security threat.

The second example illustrates the challenge of failing to cover all libraries in the whitelist (**Challenge 2**), The automated model (Chronos) identifies *io.ifar.kafka-archiver:kafka-archiver* as the vulnerable library of CVE-2024-47876, while the actual vulnerable library is *org.apache.kafka:kafka-clients*. The model depends on a large library pool, which does not include *org.apache.kafka:kafka-clients*. Therefore, in theory, the model can not correctly identify the vulnerable library of CVE-2024-47876. The model has to select the most likely library, *i.e.*, *io.ifar.kafka-archiver:kafka-archiver*, from the library pool as the vulnerable library, even though the prediction score for this library is low. If an organization depends on *org.apache.kafka:kafka-clients*, it will not receive an alert and will remain affected by this vulnerability. Due to changes in the name of the library and the emergence of new libraries, it is difficult for a library pool to cover all software libraries. As a result, the effectiveness of automated methods relying on a library pool is limited.

To address these two challenges of securing self-managed third-party libraries, automated methods should employ a library pool that is not excessively large while encompassing all libraries used by organizations. Therefore, we augment identification methods with whitelists and design filtering strategies to identify as many overlooked vulnerable libraries as possible while maintaining a low false alarm ratio.

IV. Framework: LibGuard

A. Overview

We propose LibGuard, a framework for securing self-managed third-party libraries for organizations. Figure 2 shows an overview of LibGuard. LibGuard consists of three components: *identification of vulnerable libraries* (C1), *supply of library whitelists* (C2), and *filtering based on thresholds* (C3). C1 identifies vulnerable libraries from vulnerability reports. C2 limits the scope of identification methods to the whitelist, reducing the task difficulty and improving the accuracy. C3 filters the previous output based on a threshold, identifying vulnerable libraries overlooked by organizations and controlling false alarms. The three components work synergistically to support organizations in self-managing the security of third-part libraries they depend on. LibGuard is implemented in two ways. The first is to attach library whitelists after identification methods and apply filtering based on thresholds of the similarity score. The second is to integrate library whitelists into identification methods and apply filtering based on thresholds of the model's prediction score.

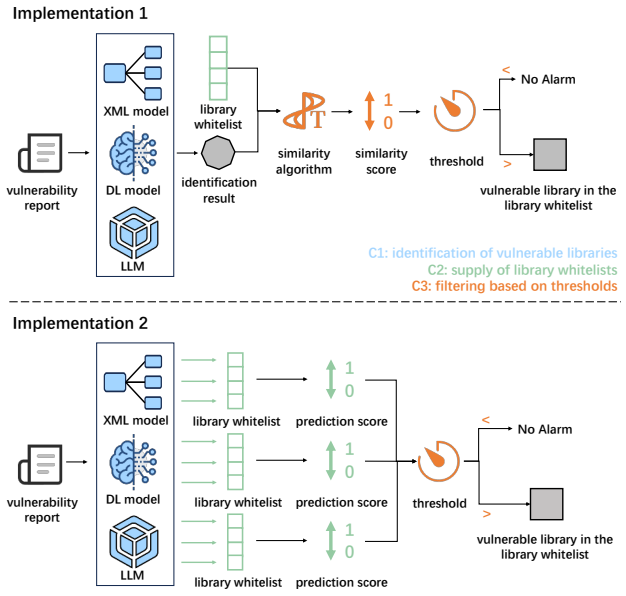

Fig. 2. An overview of LibGuard (two implementations)

B. Identification of Vulnerable Libraries (C1)

LibGuard selects three categories of methods to identify vulnerable libraries from vulnerability reports.

1) XML-based Method: We select Chronos as the XML-based method in LibGuard. The input of Chronos has not only vulnerability reports, but also the content from reference web pages. Therefore, LibGuard crawls content from frequently occurring domains to supplement Chronos input. Chronos uses the vulnerable libraries (labels) of the collected vulnerability reports to construct its mapping space and then trains a ZestXML model to learn the relationship between the vulnerability reports and the predefined mapping space (collected vulnerable libraries).

2) DL-based Method: We select VulLibMiner as the DL-based method in LibGuard. VulLibMiner requires the description of the library for data augmentation, so LibGuard collects descriptions for all Maven libraries. VulLibMiner first filters out the top-k candidate libraries from all Maven libraries using a weighted TF-IDF matching method. Then, the k candidate libraries are fed into the BERT-FNN model, which calculates the coherence score between each candidate library and the vulnerability report. The libraries with the highest scores are identified as the vulnerable libraries.

3) LLM-based Method: We select VulLibGen as the LLM-based method in LibGuard. VulLibGen requires the description of the library and the outputs of VulLibMiner (top-k libraries with the highest scores) for data augmentation. Then VulLibGen fine-tunes Vicuna-13B locally to generate the names of vulnerable libraries directly based on the vulnerability reports, the top-k candidate libraries, and the descriptions of the k libraries. Finally, VulLibGen utilizes a local search algorithm to calculate the correlation score between the generated library and existing Maven libraries. The libraries with the highest scores are identified as the vulnerable libraries.

C. Supply of Library Whitelists (C2)

LibGuard provides two methods to supply the library whitelist. The first is to attach the library whitelist after identification methods. The second is to integrate the library whitelist into identification methods.

1) Library Whitelist Attachment: Identification methods output vulnerable libraries after analyzing vulnerability reports. However, the output libraries may not exactly match the names of libraries in the library whitelist, as the identification result may not be accurate. LibGuard solves this problem by attaching the library whitelist of an organization. LibGuard collects the libraries used by organizations to build the library whitelist. Then LibGuard directly attaches the organization's library whitelist after the identification method output for subsequent operations, *e.g.*, similarity match.

2) Library Whitelist Integration: In addition to directly attaching the library whitelist, LibGuard proposes another method that integrates the library whitelist into the identification method to reduce the difficulty of identification. For the three identification methods, LibGuard designs different integrating strategies for each.

For XML-based identification methods, XML models need to build the mapping space in advance. Chronos collects vulnerable libraries of vulnerability reports to build the mapping space. To supply the library whitelist, LibGuard collects the libraries used by organizations, then adds these libraries to the previously collected vulnerable libraries and removes duplicates, and finally builds the mapping space using both vulnerable libraries and libraries in the library whitelist.

For DL-based identification methods, deep learning models receive the filter output of weighted TF-IDF as input. The weighted TF-IDF selects the top-k candidate libraries from

all Maven libraries. To supply the library whitelist, LibGuard reduces the scope of selection from all Maven libraries to only those in the library whitelist, thereby lowering the selection difficulty and improving efficiency. LibGuard then input the *k* candidate libraries selected from the library whitelist to deep learning models and calculate the coherence score for each of them.

For LLM-based identification methods, LLMs receive the retrieval result of VulLibMiner as input. VulLibMiner retrieves top-*k* candidate libraries from all Maven libraries. To supply the library whitelist, LibGuard first limits the retrieval scope of VulLibMiner from all Maven libraries to only those in the library whitelist. LibGuard then input the candidate libraries retrieved from the library whitelist to LLMs. Next LibGuard utilizes LLMs to generate the names of vulnerable libraries. Finally, based on the LLM output, LibGuard uses a local search algorithm to calculate the score for each library in the library whitelist.

D. Filtering based on Thresholds (C3)

LibGuard provides two methods for threshold-based filtering. For library whitelist attachment, LibGuard performs filtering based on similarity, and performs filtering based on models' prediction scores, otherwise.

1) Similarity-based Filtering: LibGuard selects three popular similarity algorithms to calculate similarity. Text similarity algorithms are mainly classified into three categories: *edit-based*, *token-based*, and *sequence-based*. LibGuard selects one representative algorithm from each category to calculate the similarity between identification results and each library in the library whitelist.

LibGuard selects the Levenshtein similarity [44] as the *edit-based* algorithm. Levenshtein distance is the most commonly used *edit-based* distance, measuring the minimum number of single-character operations (insertion, deletion, or substitution) required to transform one string into another. Levenshtein similarity is calculated as 1 minus the minimum number of single-character operations needed for the transformation, divided by the maximum length of the two strings. The range of Levenshtein similarity is from 0 to 1.

LibGuard selects the cosine similarity [45] as the *token-based* algorithm. Cosine similarity is a widely used *token-based* similarity and, in our setting, the resulting similarity value itself serves as the *correlation score*. Cosine similarity quantifies the cosine of the angle between two non-zero vectors in a multidimensional space, commonly used to compare the similarity between texts, documents, and other high-dimensional data points. The cosine similarity captures the direction of the vectors rather than their magnitude. The calculation of this correlation score is shown in Equation (1), where $A \cdot B$ represents the dot product of vectors A and B, $|A|$ represents the Euclidean norm of vector A, and $|B|$ represents the Euclidean norm of vector B. In text matching, vectors A and B usually represent the term frequency vectors of texts. Since term frequencies cannot be negative, the correlation score (i.e., cosine similarity) in text similarity scenarios lies in the range $[0, 1]$.

$$\cos(\theta) = \frac{A \cdot B}{|A| \cdot |B|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \quad (1)$$

LibGuard also selects the longest common substring (LCS) similarity [46] as the *sequence-based* algorithm. The longest common substring refers to the longest contiguous substring that is identical between two strings. Here the similarity value–calculated by dividing the length of the longest common substring by the maximum length of the two strings–directly represents the *correlation score* as used in Section IV-D.

After calculating similarity scores, LibGuard uses a predefined threshold to perform filtering for the final output. The filtering algorithm is as follows.

Algorithm 1: Filtering Algorithm

```
   Data: Vulnerability Report Set: VR, Library Whitelist: WL
   Result: Vulnerable Library in the Library Whitelist: L_v
1  L_v ← ∅
2  // M is a mapping from vulnerability reports to vulnerable
     libraries
3  x_v ← M(v), v ∈ VR // x_v is the identification result for v
4  for v ∈ VR do
5      for k ∈ WL do
6          // x_k is the library name of k
7          s_k ←Similarity(x_v, x_k) // calculate similarity
8          if s_k > threshold then
9              L_v ← L_v ∪ x_k // record vulnerable library
10         end
11     end
12 end
13 return L_v
```

In line 1, LibGuard initializes L_v to an empty set, indicating that there are no vulnerability libraries in the library whitelist at the beginning. In line 3, $M(v)$ represents identifying vulnerable libraries from vulnerability reports. In Lines 4-12, LibGuard compares the identification results of each vulnerability report with each library in the library whitelist. In line 7, LibGuard calculates the similarity between the identification result and the library in the library whitelist. In lines 8-9, if the similarity score is greater than a predefined threshold, LibGuard records the vulnerable library x_k. Finally, in line 13, LibGuard outputs L_v, which records the vulnerable libraries identified from the library whitelist. If L_v is empty, LibGuard would not issue an alarm, indicating that all the libraries in the library whitelist are not affected by these vulnerabilities.

2) Prediction Score-based Filtering: For library whitelist integration, LibGuard performs filtering based on the prediction scores of the models and a predefined threshold.

For XML-based identification methods, the XML model calculates a probability score for each library in the whitelist. The probability score is between 0 and 1. LibGuard utilizes a predefined threshold between 0 and 1 to perform filtering.

When the probability score is above the threshold, LibGuard will issue an alarm that this library in the whitelist is vulnerable. Otherwise, LibGuard will not issue an alarm.

For DL-based identification methods, the deep model calculates a coherence score for each library in the whitelist. The coherence score is a real number which can be positive or negative. To constrain it within a fixed range, LibGuard normalizes the coherence scores of all libraries in the whitelist, resulting in scores within the range of 0 to 1, and then applies a filter based on a preset threshold within this range. LibGuard issues an alarm when the normalized score is greater than the threshold.

For LLM-based identification methods, the correlation score is calculated by applying a local search algorithm to compare the results of the large language model and each library in the library whitelist. The correlation score is between 0 and 1. LibGuard utilizes a predefined threshold between 0 and 1 to implement filtering. When the correlation score exceeds the threshold, LibGuard triggers an alert.

The threshold can be applied to balance identification capability and false alarms. The false alarm ratio will be high when the threshold is low, increasing the overhead of manual verification. Although the false alarm ratio will be low when the threshold is high, the identification capability will also deteriorate. Therefore, when it is necessary to discover vulnerable libraries from the whitelist as much as possible (high-security need), a low threshold can be adopted, while a high threshold can be adopted to lower the extra checking expense of organizations (low-cost need).

V. Experimental Designs

A. Research Questions

The experimental evaluation of LibGuard is structured around three research questions (RQs).

RQ1: How effective is LibGuard with whitelist attachment in securing self-managed libraries?
Motivation: The identified vulnerable libraries may be inaccurate and could differ from the library names used by organizations, resulting in false negatives. It is intuitive to measure the similarity between vulnerable libraries identified with libraries in the library whitelist and then issue an alarm based on a similarity threshold. In RQ1, we evaluate LibGuard with library whitelist attachment in securing self-managed libraries.

RQ2: How effective is LibGuard with whitelist integration in securing self-managed libraries?
Motivation: Existing methods try to identify vulnerable libraries out of all libraries. However, organizations focus on the third-party libraries they use. Therefore, it is reasonable to integrate the library whitelist into identification methods to narrow the scope of determination. In RQ2, we evaluate LibGuard with library whitelist integration in securing self-managed libraries.

RQ3: How effective is LibGuard in securing self-managed libraries when the whitelist is expanded?
Motivation: Due to the constant changes in the third parties that organizations depend on, the library whitelist is constantly updated and expanded. An ideal method should be adaptable to the growth of the library whitelist. In RQ3, we investigate the stability of LibGuard performance as the library whitelist expands.

B. Dataset

Due to no dataset suitable for addressing all research questions, we develop ours through three steps.

Step 1: Project Selection. We select projects from Google and Huawei, both maintain the third-party libraries they use based on library whitelists [35, 47]. We select Maven projects from the official GitHub repositories of Google and Huawei because Maven projects have clear configurations that allow for accurate extraction of dependent third-party libraries. We select all the open-source projects from Google and Huawei (including Huawei and HuaweiCloud), respectively, and then filter out those without clear configurations, *e.g.*, pom.xml. Finally, our dataset includes 79 Google projects and 29 Huawei projects. We collect the third-party libraries used by these projects to build the library whitelist of these two organizations. The basic information of the top 10 projects, ranked by their number of third-party libraries is presented in Table I.

TABLE I
BASIC INFORMATION OF THE SELECTED PROJECTS

Project	Organization	TPLs	Stars	Forks
fhir-data-pipes	Google	113	161	90
graphicsfuzz	Google	83	572	115
error-prone	Google	74	6.9k	746
framework-for-osdu	Google	72	19	5
depan	Google	65	91	19
guice	Google	58	13k	1.7k
huaweicloud-sdk-java-v3	Huawei	210	135	116
soda_vmware_plugin	Huawei	75	2	3
vCenter_Plugin_DME	Huawei	68	3	5
spring-cloud-huawei	Huawei	66	530	224

[1] TPLs: Third-Party Libraries.

In Table I, many popular projects, *e.g.*, *error-prone* and *guice*, depend on dozens of third-party libraries. An organization owns many projects, and the third-party libraries of each project collectively form the organization's third-party library list, *i.e.*, library whitelist. We extract third-party libraries from each selected project of Google and Huawei, followed by aggregation and deduplication, ultimately obtaining the library whitelists of the two organizations. The library whitelist of Google contains 688 third-party libraries. The library whitelist of Huawei contains 604 third-party libraries.

Step 2: Vulnerability Collection. NVD vulnerability reports do not explicitly mention vulnerable libraries, while other security vendors such as GitHub Advisory [48] associate vulnerability reports with vulnerable libraries. Therefore, we crawl 5,107 vulnerability reports and their affected libraries (6,079 libraries) from GitHub Advisory. All the vulnerable libraries of these vulnerability reports are Maven libraries.

Step 3: Label Setting. To perform the research, we need to label each vulnerability report to determine whether it affects any library in the library whitelist and which libraries

in the library whitelist are affected. The vulnerable libraries of these vulnerability reports have been collected in step 2, and we compare them with the library whitelists of Google and Huawei. 402 vulnerability reports threaten 78 libraries in the library whitelist of Google, and 376 vulnerability reports threaten 64 libraries in the library whitelist of Huawei.

C. Baselines

We select three popular methods as the experimental baselines.

Chronos [12] is based on ZestXML and has been proven more effective than other XML methods in identifying vulnerable libraries. Therefore, we select Chronos as our baseline. Chronos enhances input data by extracting information from linked web pages in vulnerability reports. We implement data enhancement in the same way as the previous work [12].

VulLibMiner [14] combines a TF-IDF matcher and a BERT-FNN model to identify vulnerable libraries. VulLibMiner outperforms FastXML and LightXML on the VeraJava and VulLib datasets [43]. Therefore, we select VulLibMiner as our baseline method. The TF-IDF matcher extracts noun and adjective tags from vulnerability descriptions and then calculates the weighted TF-IDF [49] score between the vulnerability description and the library description. We collect the descriptions of all Maven libraries and all libraries in the library whitelist.

VulLibGen [13] leverages the pre-trained knowledge of LLMs to generate the names of vulnerable libraries. VulLibGen performs best on the VulLib dataset [43]. Therefore, we select VulLibGen as our baseline method. VulLibGen identifies zero-shot libraries through input augmentation based on VulLibMiner [14]. VulLibGen designs a prompt template that incorporates both the vulnerability description and the output of VulLibMiner as input. We implement VulLibGen by fine-tuning Vicuna-13B [50] using LoRA [51]. We use the same local search techniques [52] as previous work to resolve hallucination issues.

D. Evaluation Metrics

Following previous work [12], we evaluate LibGuard and baselines using Precision (P), Recall (R), and F1-score (F1) calculated for the top-k identification results. In formula 2, v is the vulnerability report, $lib\_k(v)$ is the top-k identification result of the method, $\hat{lib}(v)$ is the vulnerable libraries of the given vulnerability report v, and WL is the library whitelist. $|\hat{lib}(v) \bigcap WL|$ is the number of the overlap between vulnerable libraries of the given vulnerability report v and libraries in the library whitelist WL. The numerators of $P@k(v)$ and $R@k(v)$ are the same, both of which are the overlap of the identified top k results, the vulnerable libraries, and the library whitelist. The denominator of $P@k(v)$ is the minimum value of k and the number of the overlap between the vulnerable libraries and the library whitelist. The denominator of $R@k(v)$ is the number of the overlap between the vulnerable libraries and the library whitelist.

$$P@k(v) = \frac{lib\_k(v) \bigcap \hat{lib}(v) \bigcap WL}{min(k, |\hat{lib}(v) \bigcap WL|)} \tag{2}$$

$$R@k(v) = \frac{lib\_k(v) \bigcap \hat{lib}(v) \bigcap WL}{|\hat{lib}(v) \bigcap WL|} \tag{3}$$

$P@k$ is the average value of $P@k(v)$ for all vulnerability reports that affect the library whitelist, *i.e.*, $|\hat{lib}(v) \bigcap WL| \neq 0$. $R@k$ is the average value of $R@k(v)$ for all vulnerability samples that affect the library whitelist. $F1@k$ is a comprehensive consideration of $P@k$ and $R@k$.

$$P@k = 1/n \sum_{v=1}^{n} P@k(v) \tag{4}$$

$$R@k = 1/n \sum_{v=1}^{n} R@k(v) \tag{5}$$

$$F1@k = 2 \times \frac{P@k \times R@k}{P@k + R@k} \tag{6}$$

We set k = 1, 2, 3 and calculate the average of the three results, which are also applied by previous works [15, 7, 12, 13]. We calculate P, R, F1 only when the vulnerability reports affect the library whitelist, *i.e.*, $|\hat{lib}(v) \bigcap WL| \neq 0$

We use FA (False Alarm ratio) to evaluate the performance of LibGuard when organizations are not affected by vulnerabilities. In formula 7, $lib\_(v)$ is the identification result of the method. $\hat{lib}(v)$ is the vulnerable libraries of the given vulnerability report v. WL is the library whitelist of the organization. When the vulnerable libraries of v are not in the library whitelist, if an alert is issued, $FA(v) = 1$; otherwise, $FA(v) = 0$. FA is the average value of $FA(v)$ for all vulnerability reports that do not affect the library whitelist, *i.e.*, $|\hat{lib}(v) \bigcap WL| = 0$.

$$FA(v) = \begin{cases} 1, & \text{if } \hat{lib}(v) \bigcap WL = \varnothing,\ lib\_(v) \bigcap WL \neq \varnothing \\ 0, & \text{if } \hat{lib}(v) \bigcap WL = \varnothing,\ lib\_(v) \bigcap WL = \varnothing \end{cases} \tag{7}$$

$$FA = 1/n \sum_{v=1}^{n} FA(v) \tag{8}$$

VI. Results and Analysis

A. Effectiveness of Whitelist Attachment (RQ1)

The results of evaluating LibGuard with library whitelist attachment are shown in Table II and Table III, respectively.

Among existing methods, Chronos exhibits the weakest effectiveness. The average F1 score of Chronos is 21.16% in Google and 02.04% in Huawei. The poor performance can be explained that Chronos predefines a library pool based on vulnerable libraries collected before, which fails to cover new vulnerable libraries and the library whitelists of Google and Huawei. VulLibMiner and VulLibGen perform significantly better than Chronos. The average F1 score of VulLibMiner

TABLE II
RESULTS OF LIBGUARD WITH SIMILARITY ALGORITHMS

Org	Method	Threshold=0.3				Threshold=0.4				Threshold=0.5				Threshold=0.6				Threshold=0.7			
		P	R	F1	FA	P	R	F1	FA	P	R	F1	FA	P	R	F1	FA	P	R	F1	FA
Google	LibGuard-C-Lev	22.69	21.56	21.83	79.54	22.14	21.03	21.29	69.69	21.89	20.79	21.04	41.57	21.56	20.45	20.71	24.19	21.39	20.29	20.54	13.03
	LibGuard-C-Cos	22.32	21.13	21.41	40.28	21.73	20.58	20.85	23.99	21.48	20.37	20.63	20.79	21.39	20.29	20.54	14.01	21.14	20.03	20.29	08.17
	LibGuard-C-LCS	22.21	21.09	21.35	50.61	21.92	20.80	21.06	34.12	21.48	20.37	20.63	23.06	21.39	20.29	20.54	13.52	21.39	20.29	20.54	11.74
	LibGuard-VLM-Lev	73.60	71.37	71.94	99.74	73.69	71.39	71.97	91.46	71.70	69.40	69.98	39.71	70.65	68.39	68.96	22.60	69.49	67.41	67.93	10.24
	LibGuard-VLM-Cos	77.16	74.86	75.45	42.61	72.66	70.50	71.07	20.61	72.90	70.79	71.36	14.75	72.29	70.23	70.79	10.69	68.66	67.09	67.52	06.22
	LibGuard-VLM-LCS	73.95	71.51	72.11	43.22	72.73	70.34	70.92	32.26	72.28	70.03	70.59	24.71	71.78	69.38	69.96	12.28	70.12	67.98	68.52	09.32
	LibGuard-VLG-Lev	77.83	75.07	75.79	99.02	77.83	75.07	75.79	84.73	77.42	74.66	75.37	34.65	76.34	73.62	74.32	18.95	74.20	71.61	72.26	07.62
	LibGuard-VLG-Cos	**82.31**	**79.48**	**80.20**	37.50	77.74	75.09	75.77	16.86	76.98	74.24	74.95	11.08	78.23	75.76	76.40	07.22	73.66	71.33	71.94	**04.39**
	LibGuard-VLG-LCS	78.80	76.01	76.73	38.86	77.78	75.03	75.74	27.35	77.05	74.37	75.05	17.89	75.93	73.25	73.93	08.61	74.03	71.40	72.07	06.16
Huawei	LibGuard-C-Lev	05.50	03.83	04.18	80.39	03.75	02.61	02.82	69.33	03.68	02.42	02.67	40.40	02.93	01.88	02.06	30.41	02.75	01.71	01.88	21.87
	LibGuard-C-Cos	03.78	02.56	02.79	44.24	03.10	02.02	02.21	31.82	02.93	01.88	02.06	27.05	02.93	01.88	02.06	24.02	01.63	01.47	01.51	22.48
	LibGuard-C-LCS	05.94	04.27	04.62	48.90	03.68	02.42	02.67	35.37	03.68	02.42	02.67	22.74	03.68	02.42	02.67	21.87	03.59	02.33	02.58	21.87
	LibGuard-VLM-Lev	61.02	58.77	59.29	99.57	59.16	56.95	57.46	89.15	57.58	55.44	55.92	70.94	55.85	53.74	54.22	39.62	54.34	52.58	52.95	21.66
	LibGuard-VLM-Cos	57.31	55.17	55.65	67.56	58.33	56.23	56.70	58.85	60.67	58.89	59.30	30.70	53.18	51.60	51.98	20.01	55.64	54.61	54.89	10.52
	LibGuard-VLM-LCS	59.55	57.07	57.64	75.02	57.37	55.12	55.60	64.01	56.58	54.32	54.80	30.46	55.11	52.90	53.37	24.50	54.26	52.47	52.89	19.18
	LibGuard-VLG-Lev	**65.59**	**62.60**	**63.34**	98.72	65.51	62.54	63.27	84.53	64.11	61.28	61.97	61.96	58.01	55.20	55.88	22.95	55.79	53.24	53.84	12.04
	LibGuard-VLG-Cos	55.77	53.26	53.85	58.55	52.23	49.83	50.40	49.49	53.00	50.60	51.17	17.45	49.56	47.62	48.10	11.37	46.69	45.33	45.71	**08.05**
	LibGuard-VLM-LCS	64.60	61.69	62.39	68.19	62.96	60.32	60.94	54.49	59.60	57.01	57.62	17.29	55.61	53.02	53.63	13.10	54.05	51.81	52.35	10.34

LibGuard-C-Lev: LibGuard employs Chronos for identification and Levenshtein similarity for filtering.
LibGuard-VLM-Cos: LibGuard employs VulLibMiner for identification and Cosine similarity for filtering.
LibGuard-VLG-LCS: LibGuard employs VulLibGen for identification and Longest Common Substring similarity for filtering.

TABLE III
RESULTS OF EXISTING METHODS

Approach	Google				Huawei			
	P	R	F	FA	P	R	F	FA
Chronos	22.06	20.90	21.16	04.93	02.76	01.89	02.04	12.93
VulLibMiner	**65.44**	**63.93**	**64.29**	03.13	**50.75**	**48.80**	**49.24**	06.06
VulLibGen	62.59	60.65	61.11	**01.47**	49.22	46.56	47.19	**03.72**

is 64.29% in Google and 49.24% in Huawei. The average F1 score of VulLibGen is 61.11% in Google and 47.19% in Huawei. The improved performance can be explained that VulLibMiner and VulLibGen have a large library pool which includes many Maven libraries. However, the performance of VulLibMiner and VulLibGen is not exceptional because the library pool contains numerous libraries, many of which have similar names, introducing noise to accurate identification. All three methods have a low false alarm ratio, most of which are below 10%.

LibGuard with similarity algorithm substantially improves the performance of VulLibMiner and VulLibGen. When the false alarm ratio is within an acceptable range (around or below 15%), the average F1 score of VulLibMiner increases from 64.29% to 71.36% in Google, from 49.24% to 54.89% in Huawei, and the average F1 score of VulLibGen increases from 61.11% to 76.40% in Google, from 47.19% to 57.62% in Huawei. The improvement is due to some incorrect identification results being close to the correct ones, allowing similarity algorithms to rectify them. However, similarity algorithms fail to improve the performance of Chronos significantly, which is because its library pool is limited, resulting in the incorrect outputs deviating significantly from the correct ones.

LibGuard built on VulLibGen has better performance than built on VulLibMiner and Chronos. LibGuard built on VulLibGen with cosine similarity achieves an F1 score of 76.40% with a false alarm ratio of 7.22% for Google at the threshold of 0.6. LibGuard built on VulLibGen with LCS similarity achieves an F1 score of 57.62% with a false alarm ratio of 17.29% for Huawei at the threshold of 0.5. In most cases, the average F1 score of LibGuard built on VulLibMiner is 2-5% lower than LibGuard built on VulLibGen when the false alarm ratio is comparable. LibGuard built on Chronos demonstrates the lowest performance. The best average F1 scores of it are only 21.83% for Google and 4.62% for Huawei. The advanced performance of VulLibGen can be attributed to its use of LLMs that have a vast amount of pre-training knowledge, making the inaccurate outputs close to the correct ones.

RQ1: LibGuard with library whitelist attachment improves the average F1 score from 56.77% to 67.01% (from 64.29% to 76.40% in Google and from 49.24% to 57.62% in Google). LibGuard built on LLM-based identification methods performs best in both Google and Huawei.

B. Effectiveness of Whitelist Integration (RQ2)

The results of evaluating LibGuard by library whitelist integration are shown in Table IV.

LibGuard with integrated whitelist improves the performance of all three methods. The average F1 score of Chronos increases from 21.16% to 44.68% in Google, from 2.04% to 30.40% in Huawei, the average F1 score of VulLibMiner increases from 64.29% to 70.11% in Google, from 49.24% to 57.81% in Huawei, and the average F1 score of VulLibGen increases from 61.11% to 75.23% in Google, from 47.19% to 61.84% in Huawei. The improvement of Chronos can be explained that its library pool is augmented by the library whitelist, supplementing vulnerable libraries that are in the library whitelist but are not in the library pool. The improvement of VulLibMiner and VulLibGen can be explained by the fact that the candidate scope for the model is reduced from all Maven libraries to the library whitelist, mitigating the noise introduced by similar but irrelevant libraries.

LibGuard built on VulLibGen has better performance than built on VulLibMiner and Chronos. LibGuard built on VulLib-

TABLE IV
RESULTS OF LIBGUARD WITH INTEGRATED LIBRARY LISTS

Org	Approach	Threshold=0.3				Threshold=0.4				Threshold=0.5				Threshold=0.6				Threshold=0.7			
		P	R	F1	FA	P	R	F1	FA	P	R	F1	FA	P	R	F1	FA	P	R	F1	FA
Google	LibGuard-C	45.63	44.34	44.68	04.06	43.63	42.47	42.79	03.01	41.36	40.30	40.60	02.31	41.13	40.13	40.41	01.40	33.26	32.26	32.52	**00.98**
	LibGuard-VLM	73.15	70.32	71.02	44.20	73.15	70.32	71.02	25.61	72.24	69.41	70.11	13.72	70.99	68.16	68.87	06.35	69.75	67.04	67.71	03.01
	LibGuard-VLG	**81.65**	**78.24**	**79.06**	57.57	79.33	76.00	76.80	25.60	77.60	74.48	75.23	14.61	70.30	67.16	67.91	10.33	68.02	65.18	65.88	05.09
Huawei	LibGuard-C	31.50	30.03	30.40	09.53	30.44	28.97	29.33	07.46	29.08	27.62	27.98	05.54	28.90	27.45	27.81	02.42	26.98	25.71	26.05	**01.47**
	LibGuard-VLM	69.64	67.14	67.72	62.85	67.86	65.41	65.97	46.92	65.72	63.46	64.00	35.19	62.37	60.22	60.72	22.63	59.50	57.29	57.81	11.69
	LibGuard-VLG	**71.17**	**67.79**	**68.56**	60.28	66.06	63.03	63.71	31.04	64.17	61.14	61.84	18.77	59.99	57.26	57.88	11.61	56.25	53.47	54.10	08.14

LibGuard-C: LibGuard employs Chronos with whitelist integration.
LibGuard-VLM: LibGuard employs VulLibMiner with whitelist integration.
LibGuard-VLG: LibGuard employs VulLibGen with whitelist integration.

TABLE V
RESULTS OF LIBGUARD WITH EXPANDED WHITELIST

Approach	Google			
	+100	+200	+500	+1000
LibGuard-C-att	20.54 (↓ 0.62)	20.54 (↓ 0.62)	20.38 (↓ 0.78)	20.18 (↓ 0.98)
LibGuard-VLM-att	71.24 (↓ 0.12)	71.02 (↓ 0.34)	69.99 (↓ 1.37)	68.91 (↓ 2.45)
LibGuard-VLG-att	74.94 (↓ 1.46)	74.42 (↓ 1.98)	73.42 (↓ 2.98)	72.32 (↓ 4.08)
LibGuard-C-int	43.90 (↓ 0.78)	44.41 (↓ 0.27)	43.44 (↓ 1.24)	43.52 (↓ 1.16)
LibGuard-VLM-int	73.69 (↑ 3.58)	74.62 (↑ 4.51)	73.23 (↑ 3.08)	69.90 (↓ 0.21)
LibGuard-VLG-int	75.38 (↑ 0.15)	74.69 (↓ 0.54)	71.18 (↓ 4.05)	73.22 (↓ 2.01)

Approach	Huawei			
	+100	+200	+500	+1000
LibGuard-C-att	01.49 (↓ 0.55)	01.49 (↓ 0.55)	01.49 (↓ 0.55)	01.47 (↓ 0.57)
LibGuard-VLM-att	52.51 (↓ 2.38)	51.94 (↓ 2.95)	52.54 (↓ 2.35)	51.92 (↓ 2.97)
LibGuard-VLG-att	57.10 (↓ 0.52)	56.83 (↓ 0.99)	56.78 (↓ 1.04)	53.55 (↓ 4.27)
LibGuard-C-int	30.29 (↓ 0.11)	30.35 (↓ 0.05)	29.04 (↓ 1.36)	26.89 (↓ 3.51)
LibGuard-VLM-int	58.40 (↑ 0.59)	58.46 (↑ 0.65)	56.62 (↓ 1.19)	54.15 (↓ 3.66)
LibGuard-VLG-int	58.42 (↓ 3.42)	59.91 (↓ 1.93)	61.00 (↓ 0.84)	59.78 (↓ 2.06)

LibGuard-C-att: LibGuard employs Chronos and whitelist attachment.
LibGuard-VLG-int: LibGuard employs VulLibGen and whitelist integration.

Gen achieves an F1 score of 75.23% with a false alarm ratio of 14.61% for Google at the threshold of 0.5, and an F1 score of 61.84% with a false alarm ratio of 18.77% for Huawei at the threshold of 0.7. When the false alarm ratio is comparable, the average F1 score of LibGuard built on VulLibMiner is 70.11% for Google, and 57.81% for Huawei. The performance of LibGuard built on Chronos is still not comparable to LibGuard built on VulLibMiner and VulLibGen. The best average F1 score of LibGuard built on Chronos for Google and Huawei are 44.68% and 30.40%, respectively. The advanced performance of VulLibGen can be attributed to the pre-training knowledge and powerful learning ability of LLMs.

RQ2: LibGuard with library whitelist integration improves the average F1 score from 56.77% to 68.54% (from 64.29% to 75.23% in Google and from 49.24% to 61.84% in Google). LibGuard built on LLM-based identification methods performs best in both Google and Huawei.

C. Impact of Whitelist Expansion (RQ3)

The results of evaluating LibGuard when the library whitelist is expanded are shown in Table V. For LibGuard with whitelist attachment, the expansion of the whitelist only needs to add new libraries to the whitelist for similarity match, while for LibGuard with whitelist integration, the expansion of the whitelist needs to retrain the model for vulnerable library identification.

LibGuard with both whitelist attachment and whitelist integration performs stably during the extension of whitelists. The performance of LibGuard declines slowly with fluctuations. For LibGuard with whitelist attachment, the F1 score drops by 1.46%, 1.98%, 2.98%, 4.08% when supplementing 100, 200, 500, 1000 libraries to the Google whitelist. The F1 score drops by 0.52%, 0.99%, 1.04%, 4.27% when supplementing 100, 200, 500, 1000 libraries to the Huawei whitelist. For LibGuard with whitelist attachment, the F1 score increases by 0.15%, and drops by 0.54%, 4.05%, 2.01% when supplementing 100, 200, 500, 1000 libraries to the Google whitelist. The F1 score drops by 3.42%, 1.93%, 0.84%, 2.06% when supplementing 100, 200, 500, 1000 libraries to the Huawei whitelist. Overall, the best performance of LibGuard decreased by 2.01% on Google and by 2.06% on Huawei when supplementing 1,000 libraries. The strong stability of LibGuard performance can be explained by that LibGuard narrows the candidate scope to a small set, even after expansion. Moreover, LibGuard employs appropriate similarity algorithms and a threshold filtering mechanism, greatly reducing the interference introduced by new libraries.

The decline in LibGuard with whitelist integration is more fluctuating than with whitelist attachment. The decline of LibGuard with whitelist integration and built on VulLibGen is 0.84% when the Huawei whitelist is expanded by 500 libraries, which is unexpectedly lower than the 3.42% decline observed when the whitelist is expanded by 100 libraries. In addition, The F1 score of LibGuard with whitelist integration and built on VulLibMiner even increases by 3.08% when the Google whitelist is expanded by 500 libraries. This fluctuation can be explained by that the libraries added in each expansion are randomly selected, and thus the occurrence of similar libraries varies. But overall, as more libraries are incorporated, the probability of encountering similar ones increases, leading to a gradual decline in performance. As a result, in most cases, the performance of LibGuard with whitelist attachment declines as the whitelist expansion. The difference between whitelist integration and whitelist attachment can be explained that outputs based on the model's prediction scores (whitelist integration) are less affected by irrelevant libraries than outputs based on similarity scores (whitelist attachment). Consequently, the performance of LibGuard with whitelist integration does not experience a sharp decline when the whitelist is expanded.

RQ3: LibGuard with both whitelist attachment and whitelist integration performs stably when the whitelist is expanded. The F1 score of LibGuard declines slowly in a fluctuating manner, with an acceptable decrease around 3% when the whitelist is expanded by 1,000 libraries.

VII. Discussion

A. Application of LibGuard

Under the guidance of ISO/IEC 29147 and ISO/IEC 30111, applying LibGuard to existing vulnerability response processes optimizes the vulnerability management mechanism of organizations. Both standards provide a comprehensive framework for vulnerability management: ISO/IEC 29147 focuses on external communication and vulnerability disclosure, while ISO/IEC 30111 outlines the internal processes for handling and resolving vulnerabilities. LibGuard strengthen these processes by automating and streamlining various stages. For ISO/IEC 29147, LibGuard improves the disclosure and communication of vulnerabilities by rapidly matching the reported vulnerabilities with the organization's library whitelist, ensuring effective communications with stakeholders. This also helps maintain public trust and transparency with organizations affected by vulnerabilities. For ISO/IEC 30111, LibGuard simplifies the internal vulnerability handling process by automating the initial assessment and prioritization of vulnerabilities, allowing security teams to focus on the vulnerabilities that pose great risks to the organization. By making adaptions to align with the organization's vulnerability management technologies, LibGuard provides effective security solutions, reducing manual labor, and facilitating timely responses. The application of LibGuard enhances the security and reliability of software products/services, providing a robust and responsive framework for third-party library vulnerability management.

LibGuard can complement existing security management tools, such as BlackDuck [53] and Snyk [54]. These tools rely on vulnerability databases maintained by security teams [55], which introduces delays in reporting certain vulnerabilities[4, 3]. In contrast, LibGuard is an automated tool that identifies affected libraries directly from vulnerability reports. Although its accuracy may be lower than that of manually maintained databases, LibGuard responds more quickly to new vulnerabilities and can be applied to automatically process a wider range of vulnerability reports. By integrating LibGuard with existing security management tools, more efficient and comprehensive protection can be achieved for the software security of organizations.

B. Limitation of LibGuard

LibGuard faces a trade-off between high identification performance and low false alarms, which limits its application scope to some extent. This problem can be mitigated by designing more suitable similarity algorithms or training automated confirmation models. Although LibGuard performs best in helping organizations self-manage the security of their dependencies, its performance does not meet the needs of some security-conscious organizations. There are two primary factors contributing to the suboptimal performance: one is the implicit and inaccurate information of vulnerable libraries in the input, and the other is the suboptimal performance of existing vulnerable library identification methods. To address these issues, LibGuard can enhance its input by collecting data related to vulnerable libraries from additional data sources. Moreover, LibGuard can continuously integrate advanced identification methods with adaptations to improve its performance. In particular, we observe that LibGuard tends to increase the false alarm rate when there are many libraries with highly similar names and when the similarity threshold is set relatively loosely. To mitigate this, it is necessary to carefully calibrate the threshold based on preliminary experiments and the specific security requirements of different organizations. Furthermore, LibGuard could provide more fine-grained association displays in its maintained component list to highlight components with excessively high name similarity, helping users to make informed decisions and reduce false alerts. In addition, LibGuard was evaluated only on Java projects. However, given the broad pretraining knowledge of LLMs, supplementing vulnerability reports that affect libraries across different programming languages to the training data could enable an LLM to identify vulnerable libraries of multiple languages. If dependency lists for other languages become available, we would be committed to extending the evaluation of LibGuard.

C. Recommendation

We recommend that organizations (especially small and medium-sized ones) use LibGuard built on LLM-based identification methods to self-manage the security of their library whitelists. The experimental results show that LibGuard built on LLM-based identification methods performs best when the library whitelist is either attached or integrated. Large language models, with vast pre-trained knowledge and powerful ability to understand and learn from textual information, identify vulnerable libraries based on the descriptive information in vulnerability reports more accurately than XML-based and DL-based identification methods. LibGuard built on LLM-based methods outputs a vulnerable library that is close to the actual vulnerable library, even in cases of inaccurate identification, thereby offering great flexibility for subsequent operations, *e.g.*, similarity match. Satisfied performance can be achieved by adjusting the threshold of similarity scores or prediction scores.

We recommend that organizations set the filtering threshold between 0.5 and 0.7 when using LibGuard. LibGuard with lower threshold has better protective performance, but generates many false alarms, which excessively increases the cost of protection. In contrast, setting the threshold too high reduces false alarms, but compromises the protective performance of LibGuard. A proper threshold should be set within the interval between the prediction/similarity scores in correct and incorrect results. According to the experimental results, when

the threshold is set to 0.4 or below, many similar libraries are misreported, therefore, the false alarm ratios are high, most of which are greater than 0.30. When the threshold is set above 0.7, many correct adjustment results are overlooked, leading to a performance drop to only around 85% of the performance when the threshold is 0.3. In comparison, setting the threshold between 0.5 and 0.7 allows LibGuard to achieve a balance between performance and cost, with an F1 score close to 0.70 and a false alarm ratio controlled below 0.15.

When organizations aim for rapid adaptation, we recommend directly comparing the output of existing vulnerable library identification methods with the libraries in the whitelist based on suitable similarity, followed by threshold filtering. This implementation does not require retraining the model and therefore incurs minor overhead. When organizations seek optimal performance, we recommend integrating the library whitelist into existing identification methods and retraining models. In summary, LibGuard with whitelist attachment provides flexible and rapid implementation, while LibGuard with whitelist integration provides optimal protection performance.

VIII. Conclusion

It is crucial for organizations to self-manage the security of third-party libraries they depend on to safeguard software security. However, existing methods have limited effectiveness because they are designed for generic vulnerable library identification and ignore the requirements of organizations that they focus on the security of their dependencies. To solve this problem, we propose LibGuard, a framework to secure self-managed libraries by efficiently interacting with disclosed vulnerability reports. LibGuard supplies vulnerable library identification methods with library whitelists and applies threshold-based filtering to meet the requirements of organizations. LibGuard improves the average F1 score by over 10% and performs most effectively when built upon LLM-based methods. When setting the threshold between 0.5 and 0.7, LibGuard achieves a balance between performance and cost. As the first investigation into helping organizations secure their library whitelists, this paper offers insights for adapting generic software vulnerability management to self-managed security measures.

IX. Acknowledgments

This work is supported by the Natural Science Foundation of Jiangsu Province (No.BK20241195), the National Natural Science Foundation of China (No.62202219, No.62302210), the Open Project of Key Laboratory of Industry and Information Technology Ministry for Software Fusion Application and Testing Verification, the Science and Technology Development Program of Two Districts in Xinjiang Province, China under Grant (No.2024LQ03004), and the Innovation Projects and Overseas Open Projects of State Key Laboratory for Novel Software Technology (Nanjing University) (ZZKT2025A12, ZZKT2025B18, ZZKT2025B20, ZZKT2025B22, KFKT2025A17, KFKT2025A19, KFKT2025A20, KFKT2024A02, KFKT2024A13, KFKT2024A14, KFKT2023A09, KFKT2023A10).

References

[1] G. Liang, Y. Wu, J. Wu, and C. Zhao, "Open source software supply chain for reliability assurance of operating systems," *International Journal of Software and Informatics*, vol. 11, pp. 217–241, 2021.

[2] S. BlackDuck, "Seventy-eight percent of companies run on open source, yet many lack formal policies to manage legal, operational, and security risk," 2015.

[3] N. Imtiaz, A. Khanom, and L. Williams, "Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1540–1560, 2022.

[4] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2021, pp. 1–11.

[5] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2022, pp. 331–340.

[6] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2020, pp. 35–45.

[7] S. A. Haryono, H. J. Kang, A. Sharma, A. Sharma, A. Santosa, A. M. Yi, and D. Lo, "Automated identification of libraries from vulnerability data: Can we do better?" in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ACM, 2022, pp. 178–189.

[8] NIST, "National vulnerability database (nvd)," 2025, accessed on 2025-03. [Online]. Available: https://nvd.nist.gov/vuln

[9] Y. Lin, Y. Li, M. Gu, H. Sun, Q. Yue, J. Hu, C. Cao, and Y. Zhang, "Vulnerability dataset construction methods applied to vulnerability detection: A survey," in *Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. IEEE, 2022, pp. 141–146.

[10] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Empirical Software Engineering*, vol. 21, pp. 2268–2297, 2016.

[11] H. KEKÜL, B. ERGEN, and H. ARSLAN, "Estimating missing security vectors in nvd database security reports," *International Journal of Engineering and Manufacturing*, vol. 12, no. 3, pp. 1–13, 2022.

[12] Y. Lyu, T. Le-Cong, H. J. Kang, R. Widyasari, Z. Zhao, X.-B. D. Le, M. Li, and D. Lo, "Chronos: Time-aware zero-shot identification of libraries from vulnerability reports," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*. IEEE, 2023, pp. 1033–1045.

[13] T. Chen, L. Li, Z. ZhuLiuchuan, Z. Li, X. Liu, G. Liang, Q. Wang, and T. Xie, "Vullibgen: Generating names of vulnerability-affected packages via a large language model," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 9767–9780.

[14] T. Chen, L. Li, B. Shan, G. Liang, D. Li, Q. Wang, and T. Xie, "Identifying vulnerable third-party libraries from textual descriptions of vulnerabilities and libraries," *arXiv preprint arXiv:2307.08206*, 2023.

[15] Y. Chen, A. E. Santosa, A. Sharma, and D. Lo, "Automated identification of libraries from vulnerability data," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2020, pp. 90–99.

[16] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 356–367.

[17] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab, "Not all dependencies are equal: An empirical study on production dependencies in npm," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2022, pp. 1–12.

[18] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, 2020.

[19] D. Frey, M. Gestin, and M. Raynal, "The synchronization power (consensus number) of access-control objects: the case of allowlist and denylist," in *Proceedings of the 37th International Symposium*

on Distributed Computing. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 21:1–21:23.

[20] J. Luszcz, "Apache struts 2: How technical and development gaps caused the equifax breach," *Network Security*, vol. 2018, no. 1, pp. 5–8, 2018.

[21] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "Centris: A precise and scalable approach for identifying modified open-source software reuse," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 860–872.

[22] X. Zhan, T. Liu, L. Fan, L. Li, S. Chen, X. Luo, and Y. Liu, "Research on third-party libraries in android apps: A taxonomy and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4181–4213, 2021.

[23] S. Woo, H. Hong, E. Choi, and H. Lee, "{MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3037–3053.

[24] S. Woo, E. Choi, H. Lee, and H. Oh, "{V1SCAN}: Discovering 1-day vulnerabilities in reused {C/C++} open-source software components using code classification techniques," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6541–6556.

[25] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 595–614.

[26] W. Kang, B. Son, and K. Heo, "Tracer: Signature-based static analysis for detecting recurring vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1695–1708.

[27] T. Hunter and G. De Vynck, "The 'most serious' security breach ever is unfolding right now. here's what you need to know," *The Washington Post*, 2021.

[28] D. Goodin, "The internet's biggest players are all affected by critical log4shell 0-day," 2021.

[29] D. Foo, J. Yeo, H. Xiao, and A. Sharma, "The dynamics of software composition analysis," *arXiv preprint arXiv:1909.00973*, 2019.

[30] B. Catabi-Kalman, "Why do organizations trust snyk to win the open source security battle," 2021.

[31] S. Wu, W. Song, K. Huang, B. Chen, and X. Peng, "Identifying affected libraries and their ecosystems for open source software vulnerabilities," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[32] S. Wu, R. Wang, K. Huang, Y. Cao, W. Song, Z. Zhou, Y. Huang, B. Chen, and X. Peng, "Vision: Identifying affected library versions for open source software vulnerabilities," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1447–1459.

[33] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 869–885.

[34] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, 2022.

[35] Google Cloud, "Google cloud security documentation," 2024. [Online]. Available: https://cloud.google.com/docs/security

[36] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 181–191.

[37] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–34, 2021.

[38] Q.-S. Phan, K.-H. Nguyen, and T. Nguyen, "The challenges of shift left static analysis," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2023, pp. 340–342.

[39] M. Sánchez-Gordón and R. Colomo-Palacios, "Security as culture: A systematic literature review of devsecops," in *Proceedings of the 42nd IEEE/ACM international conference on software engineering workshops*. IEEE, 2020, pp. 266–269.

[40] C. Weir, S. Migues, M. Ware, and L. Williams, "Infiltrating security into development: Exploring the world's largest software security study," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021, pp. 1326–1336.

[41] R. Babbar and B. Schölkopf, "Dismec: Distributed sparse machines for extreme multi-label classification," in *Proceedings of the 10th ACM International Conference on Web Search and Data Mining*. ACM, 2017, pp. 721–729.

[42] M. Wydmuch, K. Jasinska, M. Kuznetsov, R. Busa-Fekete, and K. Dembczynski, "A no-regret generalization of hierarchical softmax to extreme multi-label classification." Curran Associates, 2018.

[43] T. Chen, "Vullib dataset," 2024, accessed on 2025-03. [Online]. Available: https://github.com/q5438722/VulLibMiner

[44] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Proceedings of the Soviet physics doklady*, 1966.

[45] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[46] P. Weiner, "Linear pattern matching algorithms," in *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE, 1973, pp. 1–11.

[47] Huawei Cloud, "Huawei cloud codearts development pipeline documentation," 2024. [Online]. Available: https://www.huaweicloud.com/intl/en-us/product/devcloud.html

[48] GitHub, "Github advisory database," 2025, accessed on 2025-03. [Online]. Available: https://github.com/advisories

[49] K. Spärck Jones, "Idf term weighting and ir research lessons," *Journal of documentation*, vol. 60, no. 5, pp. 521–523, 2004.

[50] The Vicuna Team, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," 2023, (accessed on 2025-03). [Online]. Available: https://vicuna.lmsys.org

[51] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *Proceedings of the 10th International Conference on Learning Representations*. OpenReview, 2022.

[52] K. Kukich, "Techniques for automatically correcting words in text," *ACM computing surveys (CSUR)*, vol. 24, no. 4, pp. 377–439, 1992.

[53] BlackDuck, "Black duck software composition analysis," 2025, accessed on 2025-03. [Online]. Available: https://www.blackduck.com/en-us/software-composition-analysis-tools/black-duck-sca.html

[54] Snyk, "Snyk software supply chain security," 2025, accessed on 2025-03. [Online]. Available: https://snyk.io/solutions/software-supply-chain-security/

[55] ——, "Snyk vulnerability database," 2025, accessed on 2025-03. [Online]. Available: https://docs.snyk.io/scan-with-snyk/snyk-open-source/manage-vulnerabilities/snyk-vulnerability-database?utm_source=chatgpt.com