Enhancing Security in Third-Party Library Reuse - Comprehensive Detection of 1-day Vulnerability through Code Patch Analysis

Shangzhi Xu[§], Jialiang Dong[§], Weiting Cai[†], Juanru Li[‡], Arash Shaghaghi[§], Nan Sun[§], Siqi Ma[§*]
[§]The University of New South Wales
Emails: {shangzhi.xu, jialiang.dong, a.shaghaghi, nan.sun, siqi.ma}@unsw.edu.au
[†]Delft University of Technology
Email: weitingcai2020@gmail.com
[‡]Feiyu Tech
Email: romangol_t@hotmail.com

***Abstract*—Nowadays, software development progresses rapidly to incorporate new features. To facilitate such growth and provide convenience for developers when creating and updating software, reusing open-source software (i.e., third-party library reuses) has become one of the most effective and efficient methods. Unfortunately, the practice of reusing third-party libraries (TPLs) can also introduce vulnerabilities (known as 1-day vulnerabilities) because of the low maintenance of TPLs, resulting in many vulnerable versions remaining in use. If the software incorporating these TPLs fails to detect the introduced vulnerabilities and leads to delayed updates, it will exacerbate the security risks. However, the complicated code dependencies and flexibility of TPL reuses make the detection of 1-day vulnerability a challenging task. To support developers in securely reusing TPLs during software development, we design and implement VULTURE, an effective and efficient detection tool, aiming at identifying 1-day vulnerabilities that arise from the reuse of vulnerable TPLs. It first executes a database creation method, TPLFILTER, which leverages the Large Language Model (LLM) to automatically build a unique database for the targeted platform. Instead of relying on code-level similarity comparison, VULTURE employs hashing-based comparison to explore the dependencies among the collected TPLs and identify the similarities between the TPLs and the target projects. Recognizing that developers have the flexibility to reuse TPLs exactly or in a custom manner, VULTURE separately conducts version-based comparison and chunk-based analysis to capture fine-grained semantic features at the function levels. We applied VULTURE to 10 real-world projects to assess its effectiveness and efficiency in detecting 1-day vulnerabilities. VULTURE successfully identified 175 vulnerabilities from 178 reused TPLs.**

I. INTRODUCTION

As software evolves various innovative functionalities nowadays (e.g., AI-based classification, unmanned operations), it makes software increasingly complicated and challenging to develop and maintain because of intricate dependencies among its massive functions. Open-source software (OSS) reuse enables developers to integrate the functionalities faster, which facilitates more efficient development and code maintenance. Simultaneously, reusing OSS supports flexible development of features because developers can either deploy OSS as TPLs exactly or in a custom manner [46]. However, this convenience also makes it more likely for developers to unintentionally introduce vulnerabilities, known as 1-day vulnerabilities, through TPL reuse [21] [43]. Such security issues are commonly brought because 1) some TPLs may no longer be actively maintained anymore, leaving their functionalities incomplete or vulnerable to security risks, especially when vulnerabilities are exploited; 2) the decentralized nature of OSSes makes it difficult to keep track of all code changes made by contributors, complicating the enforcement of security reviews and practices when reusing the TPLs; 3) TPLs are not always developed by following the best practices, which can make them more vulnerable. For instance, MOVEit Transfer, a file transfer project, experienced a security breach due to SQL injection in June 2023, which puts all dependent software at risk of unauthorized access; thus the dependent software is required to install the patched version to safeguard their data promptly [35].

To explore 1-day vulnerabilities brought by TPL reuses, some studies [26] [28] [48] only analyze the reuses without any modifications (i.e. exact reuse) by employing similarity comparison, which identifies whether the functions from TPLs are invoked in the target program. Unfortunately, exact reuse is only a small portion of TPL reuse because of its functional restrictions during software development. The flexibility of custom TPL reuse enhances functionality customization, yet it simultaneously introduces challenges in recognizing TPL reuse and detecting 1-day vulnerabilities caused by TPL reuse. Although some existing tools, such as V1SCAN [46] and MVP [51], claimed that they are capable of analyzing custom TPL deployment, these tools can only handle simple custom reuses with minor modifications, which leaves the problem far away from being fully resolved. Making 1-day vulnerability detection more complicated, developers may fix vulnerable TPL functions in a custom way, rather than using official updates or upgrading the entire TPLs [19].

To bridge the above gaps, we propose VULTURE, a novel vulnerability detection tool, to explore the potential 1-day vul-

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240576
www.ndss-symposium.org

nerabilities brought by TPL reuses *effectively* and *efficiently*. As TPLs designed for different target platforms vary significantly [54], VULTURE first conducts a mutually promoted approach, TPLFILTER, to heuristically refine existing OSSes by selecting the TPLs appropriate for the specific target platform(s). It constructs a multifaceted database that includes all the vulnerable and patched TPLs through LLM-based commit slicing. Instead of retaining all information (e.g., project code, descriptions, fix commits) of each TPL, TPLFILTER condenses each TPL version into several independent functions, and then converts the functions into compact and numerical tuples using the LSH algorithm [25]. According to the TPL database generated by TPLFILTER, VULTURE employs a similarity-based method to detect TPL reuse within the target program, identifying both the reused TPLs and their versions.

Based on the database, VULTURE undertakes a dual TPL analysis to meticulously examine exact and custom code reuse. In particular, VULTURE first conducts version-based analysis to explore exact TPL reuses by identifying the vulnerable TPL versions from the target program. Then, it employs fine-grained reformatting through code tokenization and chunk-based analysis to identify custom TPL reuse. To preserve semantic and contextual information, VULTURE generates each chunk by conducting intra-procedural analysis to collect information on variables (i.e., values, operations, and relative positions) involved in code modifications during the transition from the vulnerable code to the patched code. By comparing the chunks with the target program, VULTURE reports whether any 1-day vulnerabilities exist and pinpoints their exact locations.

To assess the effectiveness of TPLFILTER and VULTURE, we created a benchmark by manually analyzing 68 real-world projects, labeling 200 TPL reuses, and identifying 200 vulnerable reused functions from those reuses. By integrating the database created by TPLFILTER, VULTURE successfully identified 184 vulnerable reused functions, achieving an F1 @ 95.8% while the state-of-the-art tool, V1SCAN only detected 100 vulnerable reused functions, achieving an F1 @ 66.7%. VULTURE not only outperforms the state-of-the-art academic tool but also surpasses the commercial tool. We applied VULTURE and a commercial detection tool SNYK [12] to analyze TPL reuses within 10 real-world projects on a large scale. VULTURE outperformed SNYK by identifying 175 vulnerabilities from 178 TPL reuses, while SNYK identified 111 vulnerabilities and V1SCAN only identified 13. Our findings indicate that custom adaptations of TPLs are widespread, representing about 55% of all reuses. Additionally, the majority of 1-day vulnerabilities were due to the reuse of outdated TPLs, a problem often compounded by poor maintenance and the lack of thorough fix guidance from public security advisories.

Contribution:

- *A novel approach to automatically build an extendable database tailored for TPL analysis on a designated platform.*
  We design a mutually promoted approach, TPLFILTER, that integrates keyword searching with a LLM to explore the commonly used TPLs, along with the associated vulnerabilities and patch code. All vulnerabilities and patch information are then gathered to create a TPL vulnerability database.
- *An effective 1-day vulnerability detection tool to discover the reused TPLs and any potential vulnerabilities that have been exploited.* We design a 1-day vulnerability detection tool, VULTURE, that employs locality-sensitive hashing (LSH) comparisons and a dual intra-procedural analysis to exploit target programs semantically and syntactically. It can distinguish the vulnerable TPLs by recognizing both the official patches provided by TPL owners and the custom patches created by developers.
- *A comprehensive evaluation across* TPLFILTER *and* VULTURE. We evaluate both TPLFILTER and VULTURE by comparing them with state-of-the-art tools on real-world projects. VULTURE discovered 175 vulnerabilities from 10 real-world IoT projects and provided 154 patch commits.

Availability. The source code of VULTURE and the experiment datasets are available at https://github.com/ShangzhiXu/VULTURE_Detector.

II. BACKGROUND

A. Third-party Library Reuse

Developers extensively employ TPLs to cater to the diverse requirements of different users. For example, multiple MQTT client libraries are designed to simplify the deployment and implementation for connecting remote devices such as IoT devices, while zlib is frequently employed for managing file compression and decompression.

To recognize used TPLs, state-of-the-art detectors usually operate across three phases. First, a TPL database containing whitelists of known libraries is built. These whitelists are typically generated through manual analysis and require regular updates. Given the database, detectors then collect the representative features/signatures (e.g., invoked functions [30], [48], [49], keyword tokens [37], [44], function dependencies [18], [27], [31], [42], [53]) of the OSS utilized as the target for TPL reuse detection (i.e. target program). To enhance the effectiveness of TPL reuse detection, some detectors optimize the process by eliminating redundant functions and statements [48] or defining the significance level of each function [42], [49]. The similarity score between the collected representative features/signatures and the libraries stored in the database is further calculated. Reuse is confirmed if the similarity score exceeds a pre-defined threshold. The major challenge in detecting reused TPLs is *the inconsistency of reusing TPLs within different targeted programs*.

TPL database construction. Without a universal standard, developers across different platforms (e.g., IoT firmware, mobile operating systems, Open-source software) tailor varied infrastructures to meet specific objectives. Hence, TPL providers sometimes offer specialized versions of TPLs to better adapt to specific platform(s). For instance, CocoaMQTT [2], Paho Android Service [10] and Paho MQTT C/C++ client [11] are MQTT client libraries for iOS, Android, and embedded platforms, respectively. Hence, the TPL database used for reuse matching must be:

- **Comprehensive:** TPLs that are commonly invoked within the targeted platform must be included.

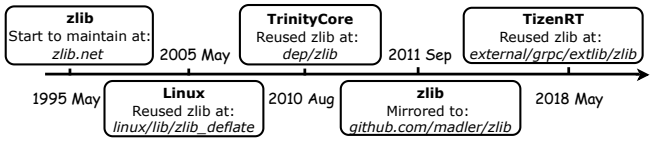

Fig. 1: Example of zlib Maintenance and Reuse

- **Specific:** Different platforms may have unique libraries that cannot be used by the other platforms. To avoid false alarms, libraries specific to other platforms, unnecessary for the targeted platform, or projects not used as libraries must be excluded during the detection process.
- **Maintainable:** TPLs are often developed to accelerate the development cycle of open-source software and are regularly updated to fix bugs and integrate new features. Hence, the libraries included in the database should also be *extendable* — to accommodate the newly-created libraries, *updatable* — to include the latest library versions, and *traceable* — to track all previous revision details.

Unfortunately, the existing TPL databases [27], [48], [49] fall short of the necessary standards as they are neither comprehensive nor specific, and they lack efficient mechanisms for updates to include additional whitelists. When updating these databases, it is imperative to reconstruct them by meticulously repeating all of the steps, including the library collection and redundancy elimination. This process is time-consuming, particularly because redundancy elimination typically incurs higher costs, requiring approximately 100 hours [48], [49]. Therefore, a powerful and adaptive database is expected to be built.

Detection of TPL reuse. Apart from the platform diversity, each TPL includes a variety of functions to serve different purposes, allowing developers to choose specific APIs that align with their unique requirements. For instance, coreMQTT [3] provides two APIs, `MQTT_ProcessLoop()` and `MQTT_ReceiveLoop()`, for receiving packets from the transport interface iteratively. If it is not necessary to keep the receiving portal alive, `MQTT_ProcessLoop()` can be selected; otherwise, `MQTT_ReceiveLoop()` should be chosen. Therefore, it is necessary to check the functions invoked in the target program by matching them with the functions declared in the libraries. Nonetheless, such similarity-based determination is heuristic and highly relies on a threshold established through manual observation. A high threshold might miss some cases; whereas a low threshold leads to many false alarms. Due to the diverse infrastructure, developers may modify some functions in TPLs to fit their code structure better, rather than reusing TPLs exactly, which complicates the similarity comparison process as well. Hence, additional details such as birth times of when each version of TPLs was created are employed as auxiliary information [48]. Relying on single auxiliary information may be insufficient to validate the complicated dependencies among TPLs and the dependent software.

Considering the real-world TPL reuses demonstrated in Figure 1 as an example, it illustrates the dependencies among zlib, Linux, TrinityCore, and TizenRT, and the birth time of each project. Specifically, zlib was created at *zlib.net* in May 1995 and was mirrored to GitHub in September 2011. Linux, TrinityCore, and TizenRT separately reuse zlib for compression and decompression; thus three projects share some common functions that are invoked from zlib. Since zlib was initially released on its own websites and only appears on GitHub later than both Linux and TrinityCore, the validation through function similarity comparison and birth time ranking [48] consequently wrongly indicates that TizenRT reuses functions from both Linux and TrinityCore.

Conclusion 1: The comparison process should be optimized by incorporating sufficient and accurate TPL information for effective TPL reuse detection.

1-day vulnerability detection within TPLs. The integration of TPLs within the development cycle of OSS not only offers convenience for implementing common features but also introduces security and privacy hazards by importing security issues. Some of these vulnerabilities arise from integrating the vulnerable versions of TPLs [16], [28], [46], [50], [58], while others occur because of the violations of specific usage requirements [19], [56], [60]. 1-day vulnerabilities are typically caused by the former.

Similar to TPL reuse detection, some vulnerable features are required for syntactical and semantic comparison to recognize the potential vulnerabilities introduced by the reused TPLs. According to the types of reused TPLs, the exact TPL reuses can be commonly analyzed via feature comparison [19], [50], [58] or even solely through the comparison of TPL versions. However, real-world TPL reuse can be customized by modifying specific segments to achieve functional objectives. While some studies have considered customized reuses, they heavily rely on coarse-grained semantic analysis to pinpoint the common features that appear in both vulnerable code and patches (e.g., variable names [28], core lines of statements [46], coarse-grained function abstraction [51]). Many false alarms are consequently reported. Hence, it is essential to differentiate the critical statements and operations that are the root causes of a vulnerability for vulnerability identification.

Considering the TPL reuses in ReactOS (shown in Figure 2) as an example, ReactOS originally reused vulnerable TPL versions of libjepg-turbo and mbedtls. Then it patches the vulnerabilities in a custom way. While the statement contents remain the same, the line numbers where the statements are located and the format of the statements changed, such alteration challenges the existing approaches [28] [46] to accurately identify the custom patches.

Conclusion 2: Selecting critical vulnerable features that contribute to the generation and mitigation of a vulnerability can enhance the effectiveness and efficiency of 1-day vulnerability detection.

B. CVE and Commit Analysis

Software assessment relies on publicly available information on vulnerabilities and security patches, which is typically

```
METHODDEF(JDIMENSION) get_8bit_row(..){
    ...
+   int cmaplen = source->cmap_length;
    ...
    for(..){
        t = GETJSAMPLE(*inptr++);
        ...
    }
```

(a) CVE-2018-14498 patch in libjepg-turbo

```
int mbedtls_x509_crt_verify_with_profile(..){
@@ -2113,7 +2115,7 @@
-       return (ret);
+       goto exit;
        ...
@@ -2125,15 +2127,21 @@
-       return (ret);
+       goto exit;
        ...
-       return (ret);
+       goto exit;
    }
```

(b) CVE-2017-14032 patch in mbedtls

```
// ReactOS reused libjpeg-turbo
METHODDEF(JDIMENSION) get_8bit_row(..){
    ...
+   int cmaplen;
    ...
+   cmaplen = source->cmap_length;
    for(...){
        t = GETJSAMPLE(*inptr++);
        ...
    }
}
...
// ReactOS reused mbedtls
int mbedtls_x509_crt_verify_with_profile(..){
@@ -2113,7 +2115,7 @@
-       return (ret);
+       goto exit;
        ...
@@ -2537,15 +2539,21 @@
-       return (ret);
+       goto exit;
        ...
-       return (ret);
+       goto exit;
    }
```

(c) Reused code in ReactOS

Fig. 2: ReactOS patching of CVE-2018-14498 and CVE-2017-14032. Due to custom reuse, the ReactOS patch differs from the official one in statement format and the line numbers where the patches are applied.

provided by CVE [5] and NVD [8]. However, many vulnerabilities disclosed by CVE/NVD come with either vague security patch information or no patch information whatsoever [45].

Commit ranking is a common process [20], [40], [41], [45] of labeling direct references between security patch-related commits and vulnerabilities. To identify the patch commit of a vulnerability, specific sets of features related to commit candidates and the vulnerabilities (e.g., vulnerability location, vulnerability identifier) are collected. Then, a pre-trained ranking model is used to estimate the relevance between each commit and the vulnerability. The most relevant commits related to security patches are supposed to be prioritized. Nonetheless, the ranking results highly rely on the quality of the selected correlation features and the information collected from CVE/NVD, which are typically manual processes. Although the manual effort is a one-time cost, the variable quality of the data collected by humans can affect the accuracy and reliability of the commit ranking. Driven by the fact that public LLMs such as GPT-4.0 have undergone extensive training on a vast array of data accessible online, including documentation, blogs, and forums, LLMs have become highly proficient in grasping the semantic meaning of natural language [13].

Conclusion 3: LLM can assist in natural language processing to analyze commits and CVE descriptions semantically and syntactically. It minimizes manual efforts, enhancing the data quality in identifying security-related information and assessing the implications of code changes.

III. Overview

the

Figure 3 illustrates the workflow of VULTURE for detecting 1-day vulnerabilities introduced by reusing TPLs. VULTURE consists of three phases, TPLFILTER *Construction*, *TPL Reuse Identification*, and *1-day Vulnerability Detection*.

TPLFILTER Construction. VULTURE employs TPLFILTER to construct a unique database tailored specifically for the target platform. It is comprised of two segments, the component segment, and the vulnerability segment. The component segment contains the TPL details (e.g., TPL names, TPL versions, and code information) while the vulnerability segment includes information on vulnerabilities that have previously or currently existed in the previous and current versions of each TPL, respectively.

TPL Reuse Identification. Given a target program, VULTURE conducts a function-level TPL reuse detection. By extracting the function hashing of all functions in the target program, VULTURE identifies the TPLs and their versions that have been reused by the target program via function-based similarity comparison. To eliminate the false alarms driven by the custom TPL reuses, VULTURE analyzes the dependencies among TPLs to optimize the results and creates a TPL reuse report.

1-day Vulnerability Detection. As TPLs are reused in two different ways, i.e., exact reuse and custom reuse, VULTURE analyzes each type of reuse by utilizing version-based matching and chunk-based analysis, respectively. Through version-based matching, VULTURE specifically identifies the functions that are reused exactly and verifies whether these functions are vulnerable by searching for the official patches provided by the TPL contributors. In cases of custom reuses, VULTURE re-generates the modified functions into chunk representatives and analyzes the code modifications within each chunk. It can determine whether the modifications introduce any vulnerabilities at a granular level. As TPLFILTER includes the official patches for certain vulnerable TPLs, VULTURE further provides fix suggestions based on the reported vulnerabilities.

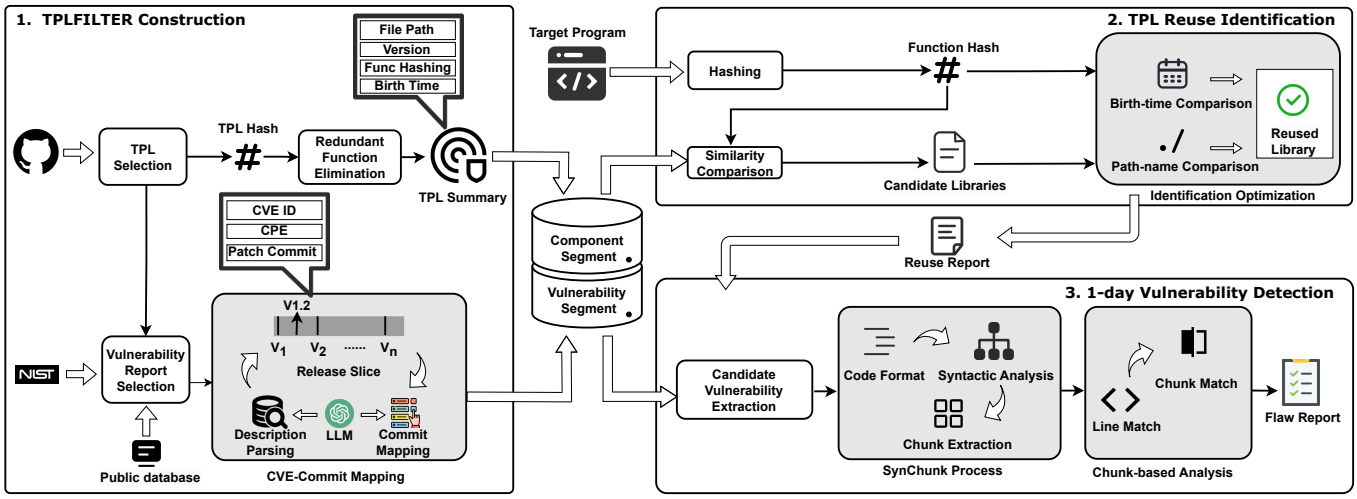

Fig. 3: Workflow of VULTURE

IV. VULTURE

A. TPLFILTER

To detect 1-day vulnerabilities introduced by TPL reuses, VULTURE must 1) identify the reused TPLs in the target program; 2) verify whether the reused TPL version is vulnerable; 3) suggest if any potential patch is available. Therefore, VULTURE is required to construct a database, containing the TPLs commonly used by a platform and the corresponding vulnerable and patch information as a detection reference.

1) TPL Selection: As TPLs vary across different platforms, VULTURE executes TPLFILTER to construct a database containing TPLs specific to the targeted platform (e.g., IoT firmware, Android, iOS). Since most open-source projects are maintained and operated on GitHub, TPLFILTER first collects all prevalent libraries from GitHub or existing TPL databases, such as OpenWRT [9] and Awesome Android [1]. To select TPLs specific to the target platform, we manually created a keyword list of the most commonly used keywords for describing the target platforms. Through the keyword list, TPLFILTER performs keyword matching to determine if the titles, tags, and descriptions of each collected TPL contain any of the specified keywords. Additionally, TPLFILTER scans the project metadata, official websites of projects, and README files to identify the keywords (e.g., system, server, firmware) for non-library project exclusion. For each selected TPL, TPLFILTER clones them from GitHub and lists all the published versions through `git tag`.

2) Component Segment Construction: Within the database, TPLFILTER gathers detailed information on each selected TPL to construct the component segment. GitHub repositories and official websites may provide a variety of information about the TPL, including details about their functions, file names, license information, and the creation time of each library (i.e., birth time). However, we observed that file names are not unique to every TPL and licenses are not formally organized by all target programs, which may negatively affect the effectiveness of TPL reuse detection. Alternatively, TPLFILTER analyzes functions declared in each TPL and their birth times to build the database, For efficiency, rather than storing all the complicated details of functions, TPLFILTER calculates a hash value for each function and stores them in the component segment.

To minimize false alarms arising from string-matching in the handling of custom patches, TPLFILTER employs LSH to process each function. LSH is a fuzzy hashing technique that hashes similar input items into the same "buckets" with high probability, thereby enhancing the accuracy of data comparison [25]. Hence, TPLFILTER analyzes each TPL version and extracts all functions with ctags. Then, it calculates the hash value of each function.

Simultaneously, it executes `git log` to identify the birth time of each function from the commit histories. Each function is then represented as $fc =< H, Birth >$, where H is the hash value of the function and $Birth$ refers to its birth time. Hence, each version of a TPL is comprised of the functions, represented as $FC = \{fc(i)|1 \leq i \leq n\}$, where n is the maximum number of functions included in the TPL version. As a TPL A may invoke functions from another TPL B (i.e., A depends on B), such dependencies may lead to redundant comparisons when analyzing function similarities.

Although some previous works (e.g., Centris [48] and OSSFP [49]) also utilize LSH to calculate hash values, these works identify redundant functions with hash value similarity comparison rather than exact match identification. Such schemes become inefficient when comparing functions on a large scale. To eliminate redundant comparisons, TPLFILTER employs a hashing-index based elimination method that narrows down the scope by searching for functions with identical hash values. Specifically, TPLFILTER first identifies the functions $fc(i)$ and $fc(j)$ by comparing hash values $H(i)$ and $H(j)$. Among all the functions with the same hash values, it selects the function with the earliest birth time and removes the other functions.

3) Vulnerability Segment Construction: To support the determination of whether the reused versions of TPLs contain any vulnerabilities, TPLFILTER collects all vulnerabilities

associated with each version of the TPLs in the component segment. As some developers may patch vulnerabilities on their own without updating TPL versions, TPLFILTER assesses security patches, instead of TPL version, to identify whether a vulnerability affects the target program. Hence, TPLFILTER collects security patches of each collected vulnerability, and compiles these vulnerabilities and their respective security patches in the vulnerability segment.

Vulnerability detail collection. As the majority of vulnerability reports are typically listed on CVE and NVD, VULTURE crawls the vulnerability reports from these websites. Within these reports, each vulnerability is assigned a unique CVE ID for identification and is described using a CPE (Common Platform Enumeration) to specify the software versions affected.

Given each TPL stored in the component segment, VULTURE first filters the vulnerability reports whose descriptions contain the TPL name at a coarse level by invoking the NVD API with `keywordSearch`[1] . Nonetheless, vulnerability descriptions encompass not only the names of vulnerable libraries but also the names of other software impacted by these vulnerable libraries, rendering keyword searches less precise. Therefore, VULTURE conducts a CPE-guided matching to filter vulnerability reports for effective vulnerability identification. Specifically, to determine whether a vulnerability affects a target TPL, it retrieves the CPE information in the vulnerability report and checks whether the target TPL name is the substring of the CPE string. If the target TPL name is matched, VULTURE regards that the vulnerability affects the target TPL and records the corresponding information of the CVE ID, its CPE, and the vulnerable TPL version(s). Typically, CPE manages the vulnerable software versions in two ways: through enumerations or version intervals. For enumeration, VULTURE directly extracts all vulnerable version numbers. In the case of interval specification, VULTURE first obtains and ranks all versions of a TPL in ascending order, and then determines the start and end versions that fall within the specific vulnerable version interval.

Unfortunately, we inspected that some vulnerability reports are informally written, thus these vulnerabilities might be overlooked by keyword searches and CPE matching. To address such issues, VULTURE extracts additional vulnerability reports from public databases including SNYK [12] and the official websites of TPLs [7] [4] for further detail matching.

Security patch collection. VULTURE collects security patches for each vulnerability to support accurate 1-day vulnerability detection and examine whether a reused TPL in the target programs has been patched. Although CVE/NVD includes patch information for some vulnerabilities, a significant portion of the patch information is not updated timely or maintained properly [6], [36]. Alternatively, TPLs maintained on GitHub frequently release security patches as GitHub commits, which typically include descriptions of vulnerabilities as well as detailed code changes.

To effectively and comprehensively obtain the vulnerability patches, VULTURE constructs an LLM-based multi-sliced patch searching approach to locate the specific security patches of each vulnerability. Given the CVE description maintained by CVE/NVD and the repository commits maintained by GitHub of each TPL, VULTURE proceeds with the following four steps:

1) LLM-based description parsing: The vulnerability detail provided by CVE/NVD typically consists of the vulnerability description (e.g., vulnerability type, vulnerable files, vulnerable functions, vulnerable variables, and some specific vulnerable features), and CPE describing the affected TPL information (e.g., vendor(s) of the TPL, TPL name and versions). Correspondingly, VULTURE employs a LLM by invoking GPT3.5 to parse the vulnerability description and then extracts the vulnerable elements affected by the vulnerability, i.e., vulnerable files, vulnerable functions, and vulnerable variables.

2) Slice-based commit filtering: As a vulnerable TPL repository may contain numerous commits that cover document changes, routine common bug fixes, feature modifications, and vulnerability patches, it is time-consuming and error-prone to analyze those commits one by one. Therefore, VULTURE conducts a date-specific commit slicing to exclude the commits confirmed to be irrelevant to the vulnerabilities at a coarse-grained level. Specifically, VULTURE first identifies all the vulnerable versions from the vulnerability description and CPE information. Since the version released immediately after the last vulnerable version is typically the first patched version of a vulnerability, VULTURE extracts the timestamps of when the last vulnerable version and the first patched version were published. Correspondingly, all the commits generated within the time range between these two timestamps are considered as potential patch commits. Then, it divides the potential patch commits into multiple slices based on average partitioning (i.e., each slice contains k commits. Within each slice i, VULTURE calculates a code diff $diff_i$ between the first and the last commits. If $diff_i$ contains any of the vulnerable elements, VULTURE labels the slice i as a candidate slice, indicating that at least one commit within this slice contains modifications related to vulnerability-specific code elements. Otherwise, it continues analyzing the next slice $i + 1$.

3) Candidate commit selection: After locating the candidate slice, VULTURE performs fine-grained analysis to select the candidate commits that are related to the vulnerability. It compares vulnerable code with each patch to generate a code diff. If the code diff includes any of the vulnerable elements, VULTURE labels the commit as a candidate commit and advances to step (4) for further confirmation.

4) LLM-based patch commit mapping: VULTURE employs LLM to confirm the patch commits. On one hand, some commits within the candidate commits may also modify the vulnerable elements yet are not the patch regarding the target vulnerability, requiring VULTURE to exclude these commits; on the other hand, some CPE information provided by NVD is incorrect [24] which may lead to false alarms. To address the challenges, VULTURE takes as input the CVE description and each candidate commit (i.e., commit description and the modified code), and invokes GPT4.0 to infer whether the CVE and the commit are relevant. It is important to note that only minor candidate commits are needed to be analyzed here, thus GPT4.0 incurs minimal costs to calculate the correlations.

A detailed example of patch commit mapping is illustrated

[1]https://services.nvd.nist.gov/rest/json/cves/2.0?keywordSearch={*}, where element $\star$ refers to the name of the target TPL.

in Appendix A.

B. TPL Reuse Identification

To determine what TPLs are reused in each target program, VULTURE takes as input the component segment of the database to check whether these TPLs are invoked. It first employs an LSH algorithm to generate a list of TPL candidates that the target program may use. Subsequently, it refines the identification process by incorporating additional information for further confirmation.

1) Candidate Library Detection: As TPLs may be reused either exactly or in a custom manner, relying solely on matching function names and code statements may overlook such custom cases. Hence, VULTURE leverages the LSH algorithm to conduct code similarity analysis.

Specifically, VULTURE first extracts all files along with their paths from the target program and then proceeds with these files one by one. Then, VULTURE divides each file within the target program into target function snippets based on the declared functions. Given each function snippet, it uses Python TLSH, an LSH-based fuzzy matching library, to generate a unique hash value, Each target function snippet i can be represented by a 2-tuple: $ft(i) =< Hash(i), Func\_path(i) >$, where $Hash$ is the hash value of the target function snippet i and $Func\_path$ represents the file path where the function snippet i is defined. VULTURE then computes the similarity between the target function snippets and TPL functions in the component segment. For each target function snippet i, VULTURE compares it with each function j from the TPL iteratively. In particular, it takes as input $Hash(i)$ and $H(j)$ and leverages TLSH to calculate a similarity score between functions i and j. If the similarity score falls below a hash threshold TH_{hash}, the function pair of i and j is considered similar. After all target function snippets are analyzed, VULTURE calculates the total number of similar function pairs. When the total number exceeds a similarity threshold TH_{sim}, it represents that the corresponding TPL may be used. Since different versions of a TPL may vary significantly, VULTURE identifies the version with the highest number of similar function pairs as a TPL prevalent version and then captures the information of the TPL (i.e., TPL name, hash value of the function in the TPL, TPL version, and the file path where the TPL is reused in the target program) to store in the candidate TPL list. It is important to note that when multiple TPL versions have the same number of similar function pairs, VULTURE randomly selects one version as the TPL prevalent version. VULTURE iteratively analyzes all functions in the target program and collects a list of candidate TPLs that are potentially reused.

2) Identification Optimization: The common similarity-based approaches, such as Centris, may produce a significant number of false positives because 1) TPLs with similar functionalities may employ the same functions; 2) the inter-dependencies among TPLs may result in a high degree of function overlap between TPLs. To accurately select which TPL is reused, VULTURE leverages auxiliary information, i.e., file paths and birth time, to optimize the candidate list.

Some existing approaches (e.g., SNYK [12]) use TPL licenses to identify reuses because the target program must declare a TPL license either in a LICENSE file or through an annotation when reusing. However, we observed that these TPL licenses are usually written inconsistently and informally, without adhering to standards. VULTURE may overlook the declared TPLs when using such information. Hence, VULTURE examines file paths illustrating where TPLs are reused in the target program and the birth time of each TPL as the indicators. Generally, the target program places code that reuses TPLs under a separate directory and then names the directory with a title similar to the TPL name. The birth time is used to pinpoint the first invocation of the reused function, thereby eliminating the redundant TPL dependencies involved during function comparison.

To confirm the reused TPLs, VULTURE first groups TPL candidates together when they share the same file path in the target program. For the TPL candidates in the same group, a path tokenizer is used to divide each file path into tokens based on the separators (e.g., space, backslash, and colon). Then VULTURE compares each token with the name of each TPL candidate and calculates a Jaccard similarity score [34], and selects the TPL candidate(s) with the highest Jaccard similarity score as the confirmed reused TPL. Note that a file may invoke functions from multiple TPLs, thus several TPLs can have the same Jaccard similarity score. VULTURE further addresses the inheritance cases where several TPLs share the same score to locate the confirmed reused TPL. First, VULTURE gathers the remaining TPL candidates sharing the same hash value H into a group, indicating that these TPLs are related. Within each group, VULTURE compares the birth time of each TPL and identifies the one with the earliest birth time as the parent TPL of the group (i.e., the confirmed TPL of the target program). All the remaining TPL candidates in the group are then removed. For each target program, VULTURE eventually generates a TPL reuse report containing all the confirmed reused TPLs.

C. 1-day Vulnerability Detection

When the TPL reuse report discloses that a vulnerable version of TPL has been reused, VULTURE then determines whether any vulnerable functions are being invoked exactly or in a custom way. Hence, VULTURE implements a dual analysis strategy at the function level, utilizing version-based and chunk-based analyses to distinguish exact and custom reuses, respectively.

1) Version-based analysis: By inspecting the reused TPLs, we found that some developers may reuse a vulnerable TPL version, but patch the vulnerable code snippets themselves. Therefore, VULTURE utilizes a diff-inspired version-based analysis to examine a TPL reuse from three aspects: 1) whether reuse is exact or customized in terms of TPL reuses; 2) whether the reused TPL version is vulnerable; and 3) whether the exploited vulnerabilities are patched.

In detail, VULTURE first takes the TPL reuse report as input to extract the reused TPL and its version. According to the TPL name and the version, it queries the vulnerability segment of the database to verify if the reused version of the TPL was linked to any CVE report. If a CVE report is identified, VULTURE extracts the vulnerable and patched version of the code related to the CVE, and then utilizes `diff` to locate the

code diff $Diff_{vp}$. Subsequently, VULTURE analyzes the $Diff_{vp}$ to determine if the modified items in $Diff_{vp}$ are functions or global declarations (e.g., global variables, structures, and macros). For functions, VULTURE extracts and records the entire function of both the vulnerable and patched code as the vulnerable and patched functions, respectively; while for global declarations, VULTURE records them as the vulnerable and patched global declarations. Then, within the target program, leveraging ctags, VULTURE specifically labels each code snippet as either a function or a global declaration and compares them with vulnerable/patched functions/declarations, respectively.

As a result, VULTURE distinguishes TPL reuses in the target program into four groups and processes each group with function-level granularity, ensuring that unused functions in the TPLs do not affect the detection accuracy:

G1: No vulnerable reuse. It indicates a target program reuses the vulnerable version of a TPL *without involving any* vulnerable functions or vulnerable global declaration.

- **Analysis:** When all the target code snippets are exactly equal to the patched function or the patched global declaration, VULTURE regards the target program as G1. The target programs in this group are considered secure.

G2: Vulnerable global declaration reuse. It represents that a target program only reuses vulnerable global declarations.

- **Analysis:** When any of the global declarations in a target code snippet exactly equals to that of the vulnerable global declarations, VULTURE regards the target program as G2.
- **Confirmation:** VULTURE employs line matching to identify the vulnerable declarations. If a global declaration that is deleted from the vulnerable TPL continues to appear in the target code snippet, VULTURE considers that a vulnerability is identified. Furthermore, when a global declaration added to the patched TPL does not appear in the target code snippet, VULTURE considers the target program as vulnerable. A 1-day vulnerability is then identified.

G3: Exact vulnerable reuse: It represents that a target program exactly reuses the vulnerable function of a TPL without providing any patches.

- **Analysis:** VULTURE labels the target program as G3 if any target functions are equal to the vulnerable functions.
- **Confirmation:** VULTURE utilizes TLSH to compute the hash values of the vulnerable function and the target function snippet. If any function in the target program shares the same hash value as the vulnerable function, VULTURE considers it unpatched and classified to G3. A 1-day vulnerability is then identified.

G4: Custom reuse: It represents that a target program reuses the vulnerable or patched functions with custom modifications.

- **Analysis:** VULTURE labels the target program as G4 if any target functions are similar to the vulnerable or patched functions. If the similarity score for any pair of functions exceeds TH_{sim}, VULTURE classifies the target program as G4.
- **Confirmation:** VULTURE uses TLSH to compare the hash values of each function in the target program with those of the vulnerable function. Since it involves custom reuse, it is uncertain whether the target program is secure; thus, VULTURE confirms the vulnerable custom reuses via a chunk-based analysis.

2) Chunk-based analysis: To preserve the semantic and syntactic information that is overlooked by function [28] and line matching [46], static analysis has traditionally been used, but it comes with significant challenges. First, static analysis requires compilable code, which is problematic because patches are often provided as isolated snippets that cannot be compiled directly. Second, static analysis demands an extensive environment setup for compilation, which is both time-consuming and resource-intensive.

To resolve the issues brought by the traditional static analysis, VULTURE utilizes chunk-based code analysis to examine custom reused functions, striking a balance between efficiency and accuracy in semantic information extraction without the reliance on compilation. For these customized functions, VULTURE verifies whether these custom modifications provide the same security functionality as those in the official patches released by TPL contributors. Specifically, for chunk-based analysis, a *chunk* represents a group of modified lines focused on a specific functionality. In a given function, the lines within the same chunk are either governed by the same control structures (e.g., `if-else`, `while`, `for`) or operate on the same set of variables. A chunk preserves both the semantic meaning and contextual relationships embedded within the patch code, ensuring an accurate assessment of custom adaptations.

Taking as input the information of CVE ID, vulnerable code snippets, patched code snippets, $Diff_{vp}$, and target code snippets, VULTURE proceeds with the following steps:

Chunk construction. When reusing TPLs, the target program might alter the format of the reused code. Thus, VULTURE first standardizes all code snippets, including target code snippets, vulnerable code, and the patched code. According to the LLVM coding standard[2], VULTURE removes unnecessary entities such as leading spaces, comments, and constant strings.

After normalization, VULTURE separately initiates code diff among the target, vulnerable, and patched code snippets. The comparison correspondingly yields three code diff (i.e., $Diff_{vt}$, $Diff_{pt}$, and $Diff_{vp}$), where $Diff_{vt}$ is the code diff between the target and vulnerable code snippets, $Diff_{pt}$ is the code diff between the target and patched code snippets, and $Diff_{vp}$ is the code diff between the vulnerable and patched code snippets. Each line in the above code diff is treated as an individual chunk. We manually predefined 40 regular expressions, covering all types of statements (e.g., call expressions, binary operations). Using these regular expressions, VULTURE then extracts the variables and operations involved in each chunk. It further adopts a disjoint-set union algorithm to identify the data-dependent chunks when 1) the chunks are part of the same control block without any nested control structures, or 2) chunks share the same variable. Eventually, all related chunks

[2]LLVM Coding Standard: https://llvm.org/docs/CodingStandards.html

are merged, while those unmerged chunks remain independent from each other.

Vulnerability detection. After constructing the chunks, VULTURE uses chunk matching to determine if variables in the target code snippet have been patched. Initially, VULTURE conducts disjoint checking by pairwisely comparing each chunk in $Diff_{vt}$ and $Diff_{pt}$ with the chunks from $Diff_{vp}$ to determine whether the patches have been applied to the target program. If both sets of $Diff_{vt}$ and $Diff_{pt}$ share the same variables with $Diff_{vp}$, VULTURE further performs line matching. A vulnerability is considered "patched" if all lines in $Diff_{vp}$ appear in $Diff_{vt}$, but disappear in $Diff_{pt}$. Otherwise, VULTURE further executes operation match. When all operations in $Diff_{vp}$ appear in $Diff_{vt}$, but disappear in $Diff_{pt}$, it indicates that the vulnerability has been patched via customized modifications. If both matching conditions are failed, a 1-day vulnerability is considered to be "found".

A detailed example for chunk-based detection is depicted in Figure 4. The code snippet A at the top represents the patch code, while code snippet B at the bottom shows the customized patch. To construct chunks, VULTURE initially treats each added line in the patch and reused code as an individual chunk. Next, VULTURE merges these chunks based on semantic relationships and context. For instance, in the patch code snippet A, the first four lines are merged into a single chunk because they share the same variable, *cmaplen*. Similarly, the last two lines are incorporated into another chunk as the same control block governs them.

After constructing the chunks, VULTURE starts to process chunk matching. In the above example, chunk C pairs with chunk E, and chunk D pairs with chunk F. Code B is classified as patched, since the line match was unsuccessful (indicated by the highlighted lines on the left side), the operation match succeeded (indicated by the highlighted lines on the right side). This scenario of custom patch may lead to false positives in other systems like V1SCAN, but VULTURE can accurately identify the patch.

Fig. 4: Chunk extraction and match

After processing all potential vulnerabilities for the target program through version-based and chunk-based analysis, VULTURE will generate a comprehensive vulnerability report. This report not only includes the CVE-ID of the vulnerability but also provides the patch commit URL from our vulnerability segment of the database, along with details of the chunks involved in the detection process, to facilitate further patching on 1-day vulnerabilities by developers.

V. EXPERIMENT

We evaluate the performance of VULTURE across two aspects: the accuracy of TPL reuse detection, and the effectiveness in detecting 1-day vulnerabilities introduced by the reused TPLs.

A. Experiment Setup

Although VULTURE can be applied to any arbitrary platform for 1-day vulnerability detection, we applied VULTURE to IoT projects as a representative in our following experiments. In detail, VULTURE collected all prevalent repositories from GitHub, OpenWRT, stm32duino, awesome-cpp, awesome-c, and mongoose-os-libs in C/C++, and then conducted TPL selection to choose the specific TPLs. After filtering the repositories via keywords, 19,057 libraries were left. These libraries were consolidated into a database named DB_{kwd}. After identifying the dependent libraries and only retaining the parent TPLs, VULTURE eventually built a TPL database DB_{iot} containing 1,872 IoT-specific TPLs, which include TPL names, all TPL versions, hash values, and birth times of functions within the TPL to store as the component segment within DB_{iot}. According to the collected TPLs, VULTURE explored 5,114 CVE reports, of which, 1,717 CVEs have GitHub patches available. These CVE reports and patches are stored as the vulnerability segment within DB_{iot}.

We evaluated the performance of each component in VULTURE against corresponding state-of-the-art works. To evaluate the database quality, (Section V-B), we selected Centris, SNYK and VFCFinder [20]. For vulnerability detection (Section V-C), we chose V1SCAN, TPLite [27], and SNYK. Although MVP is also a 1-day vulnerability detection tool, was excluded due to its lower reported performance compared to V1SCAN. Each tool was implemented by using its latest version as of April 2024, following the instructions provided by developers.

We ran VULTURE on a machine with Debian GNU/Linux 12, 32GB RAM and 1TB SSD. For patch commit collection, we set the search slice size k at 20. For similarity comparison, we set $TH_{hash} = 30$ and $TH_{sim} = 10\%$ after manual testing to achieve optimal detection results.

B. Database Evaluation

We evaluated the two segments (i.e., component segment and vulnerability segment) within the database, DB_{iot}, respectively.

1) Component Segment Evaluation: A suitable database for TPL reuse detection must be *comprehensive*, *specific*, and *maintainable* (refer to Section II); thus we conducted two experiments to evaluate whether DB_{iot} adheres to these characteristics while ensuring the database size remains relatively small:

- **Characteristic comparison.** We compared DB_{iot} with the prevalent library database published recently [48] [27].
- **Database integration.** We integrated DB_{iot} with the state-of-the-art TPL reuse detection tool, Centris, to verify if its detection performance is enhanced.

Characteristic Comparison. We compared DB_{iot} with DB_{kwd} and the databases constructed by Centris (i.e., $DB_{centris}$) and TPLite (i.e., DB_{tplite}) from three aspects: *storage consumption*, *efficiency* and *maintainability*.

The comparison results are listed in Table I. In total, DB_{iot} includes only 1,872 libraries related to IoT, requiring 3.5 GB of storage, but DB_{kwd} and $DB_{centris}$ consume ten times more storage because they include many irrelevant libraries through random TPL selection. In contrast, TPLFILTER effectively filters out most irrelevant libraries when constructing DBiot.

When processing each TPL, although DB_{iot}, DB_{kwd}, and $DB_{centris}$ cost the same time to clone one TPL repository and calculate function hash values within it, $DB_{centris}$ needs 115.1s on average to eliminate the redundant functions within each TPL whereas DB_{iot} only needs 0.1s to remove the redundancy. As most detection tools adapt similarity comparisons to identify the reused TPLs, the database accuracy significantly affects the search space and detection efficiency. Specifically, the search space of DB_{iot} is smaller than both DB_{kwd} and $DB_{centris}$, indicating that when using DB_{iot}, much fewer similarity comparisons are needed to identify one TPL reuse. With frequent updates to TPLs, databases for TPL reuse detection need to be updated and maintained easily. However, the time needed to update $DB_{centris}$ is more than 100 hours. On the contrary, DB_{iot} requires only a few minutes to update, supported by hashing-index based elimination to remove redundancy. Although TPLite also creates a database for TPL reuse detection, its database creation requires large storage to store excessive library details, including all function details. Its storage capacity became overwhelmed when only processing 1,000 TPLs, indicating that using TPLite to build a database for TPLite is impractical.

Database Integration. To test how DB_{iot} performs when integrate with state-of-the-art tools, we compared DB_{iot}, DB_{kwd} and $DB_{centris}$ in detecting TPL reuses on the Top-10 IoT projects in C/C++ from GitHub as the target programs. Details of these projects are listed in Appendix B.

Table II shows the results when integrating Centris with the three different databases. When using $DB_{centris}$, Centris only identified 71 TPL reuses; however, its performance improved significantly, detecting 174 and 112 TPL reuses when leveraging DB_{iot} and DB_{kwd}, respectively. Notably, although DB_{iot} has the smallest size, its comprehensiveness and specification enabled Centris to detect more reuses than the other two larger databases.

2) Vulnerability Segment Evaluation: We compared VULTURE against the state-of-the-art CVE patch mapping tool VFCFinder, commonly employed schemes Link Matching (directly collect patch information from CVE/NVD references) [46] and SNYK Database Retrieval [12], assessing their mapping accuracy, time cost, and space cost, respectively. The patch commits can be directly collected from CVE/NVD and are assumed available for all the solutions.

We exclude PatchScout [41] and VCMatch [45] from the comparison because both schemes demonstrably underperform compared to VFCFinder [20], as discussed and evaluated in the study.

Commit benchmark. We randomly selected 200 CVEs and manually identified the corresponding patch commits using information from the NVD website and project commits. Overall, 158 (out of 200) CVEs have been labeled with the corresponding patch commits and the other 42 (out of 200) CVEs are labeled with "None" whose patch commits cannot be found within the given CPE scope manually (from the last vulnerable version to the fixed version).

Mapping accuracy. We quantified the number of patch commits identified by each of the three methods and manually validated the results. Table III presents the number of patch commits identified by each method (**Detected**) and the number of correctly identified patches confirmed through manual verification (**Confirmed**). Additionally, to quantify the overall performance of each method, we also considered their ability to handle cases labeled with "None" and calculated the F1 score (**F1**). As shown in the table, VULTURE successfully identified the highest number of validated patch commits (124) while generating **zero** false alarms. For the 34 patch commits that were not identified, we found that 33 were due to the absence of any information related to the vulnerable elements (files, functions, variables) in the CVE descriptions for revealing vulnerability details. For instance, the description of CVE-2007-0457 [3] in the NVD is "*Unspecified vulnerability in the IEEE 802.11 dissector ...*", which provides no useful information about the vulnerable elements. The one remaining failed case occurred because the LLM (GPT-4) was unable to accurately identify the patch commit among the candidates in Step 4 (IV-A3). Instances of this kind of failure are relatively rare. In general, in terms of the F1 score, VULTURE achieved the highest F1 score among all approaches with a significant advantage.

To prevent data bias and ensure even data distribution, we divided the dataset into five groups and performed 5-fold group validation to verify the performance of VULTURE. VULTURE achieved F1 scores of 87.5%, 88.9%, 88.51%, 88.50%, and 86.11% in each validation round, respectively, demonstrating its effectiveness.

Time and space cost. We compared the time and space overhead of VULTURE and VFCFinder. To evaluate the time overhead performance, we randomly selected 50 CVEs that all the two methods could successfully identify the patch commits. For space overhead, we calculated the average space expenditure required to match each CVE directly on the ground truth dataset. The experimental results are shown in Table IV. As indicated, VULTURE exhibited the lowest time overhead (in both average and median values) and the smallest space overhead. In terms of space overhead, VFCFinder requires cloning the GitHub repository locally to obtain candidate commits, which results in significant additional space overhead. In contrast, VULTURE accesses candidate commits via the GitHub

[3] https://nvd.nist.gov/vuln/detail/CVE-2007-0457

TABLE I: Maintainability of Each Database

Database	TPL Number	Time Cost Per TPL Processing		Storage Total(GB)	Frequency of Comparisons	Detection Time	
		Clone (s)	Elimination (s)			Exact (s)	Similarity
DB_{iot}	1,872	50.5	0.1	3.5	9,207.1	2.2	256.4
DB_{kwd}	19,057	50.5	0.1	37.4	10,657.5	5.1	301.7
$DB_{centris}$	10,288	50.5	115.1	20.0	1,508,924.7	4.4	-

TABLE II: TPL Reuse Detection Result of Centris

Target	$DB_{centris}$		DB_{iot}		DB_{kwd}	
	Dtc	Cfm	Dtc	Cfm	Dtc	Cfm
AliOS-Things	22	15	81	36	78	21
LiteOS	15	6	26	9	23	4
Tasmota	28	17	93	55	85	41
TizenRT	31	22	72	37	78	22
kamailio	3	1	9	2	11	2
mbed-os	7	3	22	10	35	11
openthread	2	2	5	2	7	1
Sming	5	3	48	18	10	5
TDengine	3	1	4	3	9	4
zephyr	1	1	3	2	8	1
Total	**117**	**71**	**363**	**174**	**334**	**112**

Dtc: Reuses detected.
Cfm: Reuses confirmed with manual check.

TABLE III: Performance of Security Patch Mapping

Scheme	Detected	Confirmed	F1 (%)
Link Matching	47	47	45.86
Snyk Database Retrieval	60	60	55.05
VFCFinder (Top-1)	158	79	61.69
VFCFinder (Top-5)	158	98	71.53
VULTURE	124	124	87.94

TABLE IV: Time and Space Cost of Security Patch Mapping

Scheme	Time Cost (s)		Space Cost (MB)
	Mean	Median	Mean
VFCFinder	285.92	71.90	337.94
VULTURE	84.68	42.31	0.00

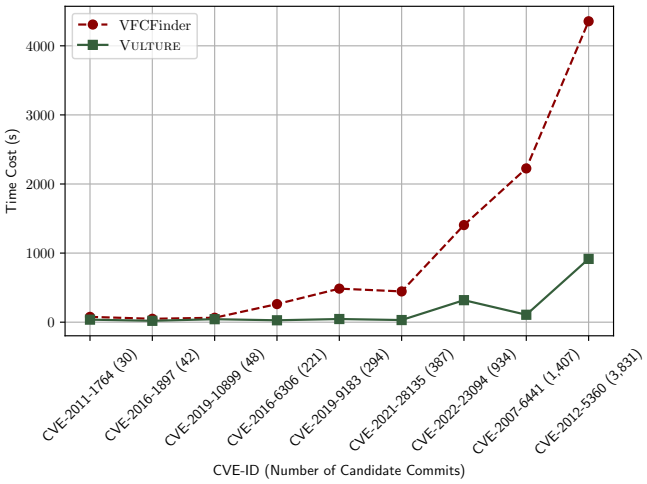

Fig. 5: Correlation between the number of candidate commits and the time cost

API and only needs to use commits at runtime as temporary variables, incurring no additional space overhead.

The Impact of Search Space on Time Cost. The time cost required for identifying patch commits is directly related to the number of candidate commits due to the difference in search space. We developed an empirical study on the 158 CVEs which are labeled with patch commits in the ground truth dataset. This study aimed to reveal how the time cost of different methods varies with the distribution of the number of candidate commits.

Here, we categorized the distribution of candidate commit numbers N into three ranges: ❶ $100 > N > 0$, ❷ $1000 > N \geq 100$, and ❸ $N \geq 1000$. Among the total 158 labeled CVEs, the distribution of the number of commits is as follows: ❶ 96, ❷ 54, and ❸ 8. Furthermore, we selected a total of 9 CVEs covering the three candidate commit number ranges to analyze the time cost of different approaches and evaluate their usability. As shown in Figure 5, the horizontal axis represents the CVE ID with the number of candidate commits in the brackets; the vertical axis denotes the time overhead, measured in seconds. As observed, the time overhead of each approach positively correlates with the number of candidate commits. Besides, with the number of candidate commits increases, the time overhead of VFCFinder becomes substantially large; in contrast, even when the number of candidate commits reaches thousands, VULTURE can still keep the time overhead within a few hundred seconds.

C. Benchmark Vulnerability Detection

We further assessed how VULTURE performs when detecting 1-day vulnerabilities introduced by the reused TPLs. We compared VULTURE with the state-of-the-art work, V1SCAN. The other tools were excluded because of various constraints. In detail, OSSFP is not available as open-source, and SNYK requires information on copyright and license, which is incompatible with our benchmark.

Vulnerability benchmark. To the best of our knowledge, there is no existing benchmark specifically for 1-day vulnerabilities in C/C++ programs. Therefore, we manually constructed a ground truth vulnerability benchmark. The benchmark comprises 200 vulnerable function reuse cases sourced from various repositories with more than 100 stars on GitHub. These 200 cases cover the reuse of 66 CVEs. Among these cases, 45% have patched the CVE they reused, while 55% have not. Additionally, 64% of these cases involve custom reuse,

TABLE V: Vulnerability Detection Result on Ground Truth

Scheme	Item	Reuse Type		Total
		Custom Reuse	Exact Reuse	
VULTURE	Dtc-P	72	19	91
	Cfm-P	65	19	84
	Dtc-N	87	22	109
	Cfm-N	78	22	100
V1SCAN	Dtc-P	51	10	61
	Cfm-P	35	10	45
	Dtc-N	66	14	80
	Cfm-N	42	13	55

Dtc: Vulnerabilities been detected.
Cfm: Vulnerabilities been confirmed with manual check.
P: Results on patched vulnerabilities.
N: Results on non-patched vulnerabilities.

TABLE VI: TPL Reuse Detection Result of VULTURE

Target	DB_{iot}			DB_{kwd}		
	Dtc	Cfm	P	Dtc	Cfm	P
AliOS-Things	47	33	0.70	61	20	0.33
LiteOS	18	12	0.66	19	4	0.21
Tasmota	66	54	0.82	63	39	0.62
TizenRT	44	34	0.77	51	20	0.39
kamailio	7	2	0.29	10	2	0.20
mbed-os	17	12	0.70	28	10	0.36
openthread	4	2	0.50	4	1	0.25
Sming	32	24	0.75	3	3	1.00
TDengine	3	3	1.00	4	2	0.50
zephyr	2	2	1.00	5	1	0.20
Total	**240**	**178**	**0.74**	**248**	**102**	**0.41**

Dtc: Reuses detected.
Cfm: Reuses confirmed with manual check.

including function name modification, statement modification, and operation modification.

Scrutinize of results. The results are shown in Table V. We compared VULTURE with V1SCAN. For custom reuse, VULTURE successfully identified 65 patched and 78 unpatched cases, with only four false negatives and 12 false positives. In contrast, V1SCAN identified 35 patched and 42 unpatched cases. For exact reuse, VULTURE identified 19 patched and 22 unpatched cases without any false negatives or false positives, whereas V1SCAN detected 10 patched and 13 unpatched cases with one false positive. Overall, VULTURE achieved a 95.8% F1 score, significantly outperforming V1SCAN, which achieved only a 66.7% F1 score.

After manually being confirmed by three PhD students from computer science majors, we observed that V1SCAN produced a significant number of false negatives due to the limited comprehensiveness of its database. Additionally, it generated many false positives, which stemmed from inherent design flaws in its detection algorithm, as detailed in two motivational examples. VULTURE reported 4 false negatives and 12 false positives in total. Out of these, 4 false negatives and 8 false positives were a result of the limitations inherent in chunk-based detection methods. Chunk-based method allows us to pinpoint modified code lines only within their respective control statements. If a modified line is not associated with any control statement, its precise location remains unidentifiable, which can result in inaccuracies. Additionally, the other 4 false positives arose from the limitations of similarity-based detection. If the reused function is significantly modified, the $Hash(i)$ and $H(j)$ can't be identified accurately (IV-B1), leading VULTURE to miss recognizing the actual reused function. Nevertheless, these issues are relatively infrequent in real-world scenarios.

D. Reuse and Vulnerability Detection In the Wild

To ensure the robustness of our evaluation, we tested how does VULTURE performs when facing real-world software. The target programs selected for this analysis are consistent with those listed in Table II. We then conducted a comprehensive analysis involving both TPL reuse detection and 1-day vulnerability detection using VULTURE.

1) TPL reuse detection: In this experiment, we executed VULTURE to detect the reused TPLs in the target programs by utilizing DB_{iot} and DB_{kwd}, respectively.

Table VI demonstrates the detection results. VULTURE correctly detected 178 reused TPLs when using DB_{iot}, achieving a precision of 74%; however its precision is only 41% when integrating DB_{kwd}. Upon manually inspecting the failed cases, we found that 18 out of 62 were caused by the non-library projects in the DB_{iot}. These projects, such as LuatOS, are not libraries themselves but are still reused by many other software, making them difficult to filter out manually. These non-library projects affect the specification of the database and cause VULTURE to mistakenly identify them as the reused TPL instead of the actual parent libraries. Nine failed cases are caused by 9 TPLs which are filtered out by VULTURE because these libraries are not prevalent on GitHub with a few stars (< 100 stars). Due to the preset thresholds, TH_{hash} and TH_{sim}, 29 cases failed as a result of overestimating similarity. Unfortunately, adjusting the thresholds could potentially result in additional failed or missed cases. The final six failed cases were due to the reuse of prevalent functions by many TPLs, which prevented VULTURE from accurately identifying the actual TPL being reused. Such prevalent functions can be categorized into cryptographic functions (e.g., `md5Update`, `parse_hex4`), utility functions (e.g., `hammingDistance`), and common functions (e.g., `strcpy`).

The disparity in the detection results between using DB_{iot} and DB_{kwd} arises because DB_{kwd} contains many non-IoT-specific libraries. This lack of specificity results in incorrect identification of parent TPL reuse as child TPL reuse, leading to a high number of false positives. In contrast, DB_{iot}, as a curated subset focused on IoT-related libraries, minimizes such misidentifications and achieves higher specificity. We did not incorporate VULTURE with $DB_{centris}$ due to its lack of critical information (e.g., birth time, file path information) required by VULTURE.

2) 1-day vulnerability detection: Here, we applied VULTURE, V1SCAN and SNYK to identify real-world 1-day vulnerabilities. Table VII only highlights the top 5 target programs with the highest number of detected vulnerabilities. The remaining 5 target programs were found to have no vulnerabilities according to the detection results.

Table VII shows that VULTURE is the most effective tool, identifying 175 vulnerabilities, and consistently outperforms SNYK and V1SCAN across all test targets. In comparison,

TABLE VII: Vulnerability Detection Result in Wild Software

Target	VULTURE		SNYK		V1SCAN	
	Dtc	Cfm	Dtc	Confm	Dtc	Cfm
AliOS-Things	93	89	105	84	8	2
LiteOS	19	19	22	16	3	3
TizenRT	68	66	16	10	11	8
Tasmota	1	1	2	0	0	0
TDengine	0	0	3	1	0	0
Total	**181**	**175**	**148**	**111**	**22**	**13**

Dtc: Vulnerabilities been detected.
Cfm: Vulnerabilities been confirmed with manual check.

TABLE VIII: Time cost of TPL reuse and 1-day vulnerability detection across different tools (in seconds)

Target	VULTURE		V1SCAN	
	TPL reuse	1-day	TPL reuse	1-day
AliOS-Things	20.1	3.0	23.5	11.8
LiteOS	14.1	3.9	28.1	8.7
Tasmota	5.5	129.9	7.1	-
TizenRT	9.2	2.1	8.1	8.8
TDengine	37.2	-	59.4	-

The values in the table represent the average time (in seconds) required to detect a single TPL reuse or a single 1-day vulnerability. A dash ("-") indicates that no reuses or vulnerabilities were identified.

SNYK detected 111 vulnerabilities, and V1SCAN detected only 13. Furthermore, VULTURE also pinpointed the exact locations and specific code statements that required to be patched, and provided 154 GitHub patch commit URLs for the identified vulnerabilities. However, SNYK only reported the CVE ID and provided 59 GitHub patch commit URLs, without offering further detailed information about the specific code requiring patching, and V1SCAN provided only 13 GitHub patch commits and limited information about vulnerabilities, such as the vulnerable file and function names,

While manually verifying the missed cases, we observed that SNYK heavily relies on version information maintained by the target program. When a program does not maintain such information well, such as TizenRT, SNYK will miss numerous vulnerabilities. When analyzing 87 vulnerabilities missed by V1SCAN, we found that 64 were caused by its limited database and others resulted from its coarse-grained line-matching approach. Besides, SNYK and V1SCAN incorrectly reported 37 and 9 vulnerabilities, respectively. Our manual checks revealed that these inaccuracies stemmed from SNYK's reliance on version-based detection and V1SCAN's coarse-grained line-matching methods.

While analyzing the six vulnerabilities missed by VULTURE, but detected by SNYK and V1SCAN, three of them were missed due to the absence of patch commits which are mandatory for VULTURE to locate patches. The remaining cases involved extensive code modifications, which caused VULTURE to fail in pairing the reused functions.

3) Time cost: We assessed the time cost of VULTURE and V1SCAN in detecting TPL reuse and 1-day vulnerabilities. Table VIII illustrates the time cost. Typically, VULTURE costs no more than 25s to recognize all TPL reuses and less than 5s to identify a 1-day vulnerability even when handling large programs like AliOS-Things. In contrast. V1SCAN usually takes more than 25s to locate a reuse and more than 5s to report a 1-day vulnerability. When analyzing programs with extensive TPL reuses (e.g., Tasmota), VULTURE takes longer to identify a single 1-day vulnerability, but V1SCAN times out without reporting any.

E. Limitation

Database Accuracy. The performance of VULTURE heavily relies on the database quality. Since VULTURE uses vulnerable elements and patched code to identify 1-day vulnerabilities, poor quality or inconsistent formats in the data provided by NVD significantly impact the performance of VULTURE. Furthermore, VULTURE utilizes LLM to capture vulnerable elements for patch identification, which makes VULTURE highly dependent on the performance of LLM. Although LLM performs well generally when processing natural language contexts, manual verification might still be necessary to ensure the accuracy of patch mapping. Otherwise, the performance of the vulnerability detection can be negatively affected.

Similarity Comparison Limitation. Although VULTURE avoids using function or line matching to identify custom reuses, the employed LSH comparison can still bring inaccuracies when processing custom reuses that involve extensive modifications. The settings of similarity thresholds TH_{hash} need to be adjusted to match various databases and accommodate different application scenarios.

Dataflow Restrictions in Chunk-based Analysis. As VULTURE generates chunks by analyzing code diff and the involved variables, it may cause incomplete data flows within chunks, leading to certain information loss. Consequently, VULTURE performs poorly when analyzing target programs with extensive code modifications. In addition, the accuracy of chunk generation also depends on the recognition results of vulnerable elements extracted from each CVE description. Correspondingly, chunk-based analysis is also affected the capability of LLM in processing natural language descriptions.

VI. RELATED WORK

Sections A, B, and C present the related work on *TPL Reuse Detection*, *Security Patch Collection*, and *TPL Vulnerability Detection*, respectively.

A. TPL Reuse Detection

TPL reuse detection is an essential task aimed at identifying the TPLs that software relies on, thereby facilitating comprehensive software maintenance and management [32]. Several detection approaches [14], [23], [27], [30], [33], [37], [44], [48], [49], [58] targeting distinct platforms have been proposed.

TPL reuse tends to introduce similar or even identical code segments from TPL into software, based on this, some studies proposed keyword token-based code reuse detection. SourcererCC [37] leveraged token-based detection, which matches similar blocks using a bag-of-tokens-based strategy. Further, CCAligner [44] introduced the concept of a code window and additionally considered edit distance to detect large-gap code reuse. However, token-based methods possessed poor

performance in detecting customized TPL reuse, and also cannot handle the issue of nested TPLs that interactive code segments. Targeting the above issues, function-level code reuse detection schemes are developed. Woo et al. [48] proposed Centris to match unique parts of TPLs by hashing functions and eliminate duplicated code to extract function features. Similarly, Wu et al. [49] developed OSSFP which focuses on identifying core functions. However, Jiang et al. [27] pointed out that in practical deployment, Centris still exhibits poor performance in cases of TPL nesting. To address this, TPLite [27] was developed which introduced function birth time and the directory they locate in to build a dependency graph and analyze the nesting relationships.

Nevertheless, the aforementioned methods require prior analysis of dependencies between TPLs before detection, which means the candidate TPL database cannot be easily expanded. Adding any new TPL necessitates reanalyzing all dependencies — an extremely time-consuming process. Therefore, as TPL pool and software complexity increasingly develop, TPL reuse detection tools must be maintainable, which is compatible in VULTURE.

B. Security Patch Collection

To collect security patches, existing solutions include retrieving patch information from vulnerability maintenance platforms [15], [46], [52] and mapping GitHub commits related to vulnerabilities from GitHub repositories [20], [38], [41], [45]. Some studies [15], [46], retrieved patch links by accessing the CVE/NVD websites and extracting the "Hyperlink" field. However, due to the untimely updates of CVE/NVD, the number of patches collected through this method is quite limited. Distinctively, Tracer [52] collected security patches from multiple known sources, such as Debian and Red Hat, together with CVE/NVD. However, the issue of untimely updates is prevalent across all the platforms.

To address the aforementioned issues, other works [20], [38], [41], [45], pursued an effort to identify patches from GitHub commits. Tan et al. [41] introduced PatchScout, which analyzed the correlation between vulnerability information and GitHub commits through a ranking strategy. Similarly, [20], [38], [45] also developed ranking-based methods, incorporating machine learning and deep learning models (e.g., XGBoost [17], CNN [29], CodeBERT [22]) to extract features. However, ranking-based solutions cannot precisely map patch commits, as depicted in the studies, highly recall rates can only be achieved in Top-N (e.g., Top-5) scenarios, which necessitates substantial manual efforts for verification. Moreover, these schemes cannot cope with the absence of patch commits for some CVEs. If the CPE provided on CVE/NVD is wrong or the patch is just released by other channels, false alarms would be returned with recognizing commits possessing some degree of similarity.

Differently, VULTURE filters candidate commits based on affected code elements, ensuring the exclusion of non-patch commits; beyond that, VULTURE leverages LLM to logically determine the actual patch commits, effectively addressing the defects of existing solutions.

C. TPL Vulnerability Detection

TPL vulnerability detection primarily relies on TPL reuse detection, associated with TPL vulnerability analysis. ReDeBug [26] employed a token-based approach for detecting reused vulnerable code through token-level similarity comparison. Similarly, Kim et al. [28] proposed a high-efficiency TPL vulnerability detection strategy VUDDY based on function comparison, with constructing hash-based function fingerprints. Zhang et al. [57] designed FIBER, leveraging subgraph matching to trace the propagation of patches. However, these methods struggle to detect vulnerabilities in cases of complex custom TPL reuse, limiting their practicality in real-world software. To address the above issues, Xiao et al. [51] proposed MVP to extract coarse-grained semantic information for identifying modified vulnerable code clones. Woo et al. [47] introduced Movery, which constructs function fingerprints towards core vulnerability features based on vulnerability-specific code and patch lines. Beyond that, V1SCAN [46] further extracted patched code lines from vulnerability patches and, when detecting TPL vulnerabilities, further reduced false alarms by comparing patch lines. Nevertheless, these methods still exhibit limitations, resulting in false positives in complicated custom reuse. In detail, Movery fails to deal with the reused vulnerable code that has been patched; while V1SCAN solely relies on counting patch lines, instead of patch contents, which is nearly tantamount to random judgment when handling some real-world software. Additionally, some studies focus on the analysis of TPL vulnerabilities in IoT firmware [58], [59], mobile software [19], [33], [50], [55] and web applications [39], offering solutions tailored to other platforms and various programming languages.

VULTURE employs static analysis to convert code to chunks based on vulnerability features. Compared to existing approaches, VULTURE can detect defects introduced by custom TPL reuse and analyze whether custom patch exists. Additionally, VULTURE's TPL database is maintainable, allowing it to adapt to the evolving software supply chain environment, a capability that existing solutions lack.

VII. CONCLUSION

This study reveals that achieving accurate software supply chain security detection requires a comprehensive toolchain. We introduced VULTURE, which employs a combination of database construction, security patch collection, TPL reuse detection, and library vulnerability detection. Through the construction of a comprehensive database and advanced algorithms, VULTURE minimizes false alarms and improves detection efficiency. Evaluation shows that VULTURE significantly outperforms state-of-the-art tools, including commercial solutions. Despite some limitations, VULTURE proves to be effective in real-world scenarios, offering developers a robust tool to mitigate risks associated with TPL reuses.

REFERENCES

[1] "awesome-android-libraries," accessed: 2024-06-03. [Online]. Available: https://github.com/wasabeef/awesome-android-libraries

[2] "Cocoamqtt," accessed: 2024-06-03. [Online]. Available: https://github.com/emqx/CocoaMQTT

[3] "coremqtt," accessed: 2024-06-03. [Online]. Available: https://freertos.org/Documentation/api-ref/coreMQTT/docs/doxygen/output/html/index.html

[4] "curl cve doc," accessed: 2024-06-03. [Online]. Available: https://curl.se/docs/security.html

[5] "Cve," accessed: 2024-06-03. [Online]. Available: https://cve.mitre.org/

[6] "Cve-2024-2398," accessed: 2024-06-03. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-2398

[7] "ffmpeg security," accessed: 2024-06-03. [Online]. Available: https://ffmpeg.org/security.html

[8] "Nvd," accessed: 2024-06-03. [Online]. Available: https://www.nist.gov/

[9] "openwrt," accessed: 2024-06-03. [Online]. Available: https://openwrt.org/packages/index/libraries

[10] "paho.mqtt.android." [Online]. Available: https://github.com/eclipse/paho.mqtt.android

[11] "paho.mqtt.embedded-c," accessed: 2024-06-03. [Online]. Available: https://github.com/eclipse/paho.mqtt.embedded-c

[12] "Snyk vulnerability database," accessed: 2024-06-03. [Online]. Available: https://security.snyk.io/

[13] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[14] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 23rd ACM SIGSAC conference on computer and communications security (CCS)*, 2016, pp. 356–367.

[15] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: Automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2021, pp. 30–39.

[16] I. Chang, K. Sotiraki, W. Chen, M. Kantarcioglu, and R. Popa, "Holmes: Efficient distribution testing for secure collaborative learning," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4823–4840.

[17] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 785–794.

[18] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "Libvdiff: Library version difference guided oss version identification in binaries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–12.

[19] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 24th ACM SIGSAC Conference on computer and communications security (CCS)*, 2017, pp. 2169–2185.

[20] T. Dunlap, E. Lin, W. Enck, and B. Reaves, "Vfcfinder: Pairing security advisories and patches," in *Proceedings of the ACM 19th ASIA Conference on Computer and Communications Security (AsiaCCS)*, 2024.

[21] Endor Labs, "State of dependency management," 2023, accessed: 2024-07-09. [Online]. Available: https://www.endorlabs.com/learn/state-of-dependency-management

[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the ACL Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 1536–1547.

[23] J. Guo, M. Zheng, Y. Zhou, H. Wang, L. Wu, X. Luo, and K. Ren, "ilibscope: Reliable third-party library detection for ios mobile apps," *arXiv preprint arXiv:2207.01837*, 2022.

[24] Y. He, Y. Wang, S. Zhu, W. Wang, Y. Zhang, Q. Li, and A. Yu, "Automatically identifying cve affected versions with patches and developer logs," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 2, pp. 905–919, 2024.

[25] O. Jafari, P. Maurya, P. Nagarkar, K. M. Islam, and C. Crushev, "A survey on locality sensitive hashing algorithms and their applications," *arXiv preprint arXiv:2102.08942*, 2021.

[26] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *IEEE 33rd Symposium on Security and Privacy (S&P)*, 2012, pp. 48–62.

[27] L. Jiang, H. Yuan, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Third-party library dependency for large-scale sca in the c/c++ ecosystem: How far are we?" in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 1383–1395.

[28] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *IEEE 38th Symposium on Security and Privacy (S&P)*, 2017, pp. 595–614.

[29] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[30] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 335–346.

[31] S. Li, Y. Wang, C. Dong, S. Yang, H. Li, H. Sun, Z. Lang, Z. Chen, W. Wang, H. Zhu *et al.*, "Libam: An area matching framework for detecting third-party libraries in binaries," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–35, 2023.

[32] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on github," in *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 1, 2017, pp. 1–28.

[33] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE-Companion)*, 2016, pp. 653–656.

[34] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of the International Multiconference of Engineers and Computer Scientists (IMECS)*, vol. 1, no. 6, 2013, pp. 380–384.

[35] Progress, "MOVEit Transfer Critical Vulnerability CVE-2023-35036," 2023, accessed: 2024-07-09. [Online]. Available: https://community.progress.com/s/article/MOVEit-Transfer-Critical-Vulnerability-CVE-2023-35036-June-9-2023

[36] A. Sabetta, S. E. Ponta, R. C. Lozoya, M. Bezzi, T. Sacchetti, M. Greco, G. Balogh, P. Hegedűs, R. Ferenc, R. Paramitha *et al.*, "Known vulnerabilities of open source projects: Where are the fixes?" *IEEE Security & Privacy*, 2024.

[37] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1157–1168.

[38] K. Shen, Y. Zhang, L. Bao, Z. Wan, Z. Li, and M. Wu, "Patchmatch: A tool for locating patches of open source project vulnerabilities," in *IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, pp. 175–179.

[39] Y. Shi, Y. Zhang, T. Bai, L. Zhang, X. Tan, and M. Yang, "Recurscan: Detecting recurring vulnerabilities in php web applications," in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 1746–1755.

[40] Z. Shi, N. Matyunin, K. Graffi, and D. Starobinski, "Uncovering cwe-cve-cpe relations with threat knowledge graphs," *ACM Transactions on Privacy and Security*, vol. 27, no. 1, pp. 1–26, 2024.

[41] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking," in *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 3282–3299.

[42] W. Tang, D. Chen, and P. Luo, "Bcfinder: A lightweight and platform-independent tool to find third-party components in binaries," in *25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 288–297.

[43] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in c/c++ ecosystem," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–12.

[44] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: A token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 1066–1077.

[45] S. Wang, Y. Zhang, L. Bao, X. Xia, and M. Wu, "Vcmatch: A ranking-based approach for automatic security patches localization for oss vulnerabilities," in *IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 589–600.

[46] 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.

[47] 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.

[48] 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 *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 860–872.

[49] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions," in *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 270–282.

[50] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "Libscan: Towards more precise third-party library identification for android applications," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3385–3402.

[51] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "MVP: Detecting vulnerabilities using Patch-Enhanced vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.

[52] C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, "Tracking patches for open source software vvlnerabilities," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 860–871.

[53] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 363–376.

[54] J. Zeng, D. Han, Y. Zhu, Y. Wang, and F. Weng, "A survey of third-party library security research in application software," *arXiv preprint arXiv:2404.17955*, 2024.

[55] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1695–1707.

[56] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 919–930.

[57] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 887–902.

[58] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu *et al.*, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 442–454.

[59] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu *et al.*, "One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in iot firmware," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[60] B. Zhao, S. Ji, X. Zhang, Y. Tian, Q. Wang, Y. Pu, C. Lyu, and R. Beyah, "Uvscan: Detecting third-party component usage violations in iot firmware," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3421–3438.

APPENDIX

A. Appendix A

In this section, we present the complete process used by VULTURE to identify the patch commit for a given vulnerability, using CVE-2013-4080 as an instance.

The CVE/NVD provides a description for CVE-2013-4080: "The dissect_r3_upstreamcommand_queryconfig function in epan/dissectors/packet-assa_r3.c in the Assa Abloy R3 dissector in Wireshark 1.8.x before 1.8.8 does not properly handle a zero-length item, which allows remote attackers to cause a denial of service (infinite loop, and CPU and memory consumption) via a crafted packet". The mapping process is detailed as follows:

1) LLM-based description parsing.

VULTURE analyzes the CVE description to parse vulnerable elements. The parsing results are as follows:

- File: packet-assa_r3.c
- Function: dissect_r3_upstreamcommand_queryconfig
- Variable: None

2) Slice-based commit filtering.

First, VULTURE performs date-specific commit slicing to pinpoint the date range which contains the patch commit. According to CVE/NVD, "wireshark-1.8.7" is the last vulnerable version and "wireshark-1.8.8" is the fixed version, narrowing the time range to (2013-05-17T16:41:42Z, 2013-06-07T15:49:07Z). A total of 348 commits fall within this period.

Next, VULTURE divides these commits into 18 slices, each containing 20 commits (the last slice contains only 8 commits). It analyzes code changes in each slice by calculating the code changes between the first and the last commit, marking those with vulnerable elements as candidate slices. Only the third slice meets this criterion.

3) Candidate commit selection.

VULTURE sequentially analyzes the 20 commits within the candidate slice, selecting the commit that genuinely modifies the vulnerable elements as the candidate commit. Only the commit with the hash 779d28d39039ada8970c910d8350fc2eb05cf00a is identified as the candidate commit in this step.

4) LLM-based patch commit mapping.

By combining the CVE description with the candidate commit, VULTURE employs feature engineering to invoke an LLM (GPT-4.0) for analyzing whether the candidate commit serves as the patch for the vulnerability. Finally, VULTURE successfully identifies the patch commit (779d28d39039ada8970c910d8350fc2eb05cf00a) for CVE-2013-4080.

B. Appendix B

The details of targeted projects used in the simulation are listed in Table IX. The "Stars" column shows the number of stars on GitHub, the "Lines" column indicates the number of

TABLE IX: Target programs details

Target Program	Stars	Lines	Version
AliOS-Things	4.5K	4.62M	a99f207
LiteOS	4.8K	1.17M	2f8fdf9
Tasmota	21.4K	1.59M	4aa2da3
TizenRT	0.5K	2.16M	db112db
kamailio	2.1K	1.16M	999d0c6
mbed-os	4.6K	8.64M	c7ea9c1
openthread	3.4K	451K	2408c89
Sming	1.4K	32.33M	895535e
TDengine	22.8K	774K	8703373
zephyr	9.7K	2.18M	2314a2c

lines of C/C++ code, and the ”Version” column contains the commit hash for each program at the time of access.