2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-5388-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISSRE62328.2024.00031

CovSBOM: Enhancing Software Bill of Materials with Integrated Code Coverage Analysis

Yunze Zhao*, Yuchen Zhang*✉, Dan Chacko†, Justin Cappos*
*New York University
†Depository Trust & Clearing Corporation

***Abstract*—The widespread integration of open-source software into commercial codebases, government systems, and critical infrastructure presents significant security challenges, particularly due to the inclusion of vulnerable components. Software Bills of Materials (SBOMs) are crucial for tracking these components; however, they lack detailed insights into the actual utilization of each component, thereby limiting their effectiveness in vulnerability management. This paper introduces CovSBOM, a novel tool that integrates code coverage analysis into SBOMs to provide enhanced transparency and facilitate precise vulnerability detection. CovSBOM addresses the gap between current SBOM and security scanning tools by providing detailed insights into which parts of third-party libraries are actually being used, thereby reducing inefficiencies and the misallocation of developer resources caused by overemphasizing irrelevant vulnerabilities. Through a comprehensive evaluation of 23 large-scale applications, encompassing 1,614 dependencies and 145 vulnerability alerts, CovSBOM has demonstrated a significant reduction in false positives, accurately identifying 105 such instances. This improvement enhances the precision of vulnerability detection by approximately 72%, while effectively maintaining a reasonable level of scalability and usability.**

***Index Terms*—SBOMs, Software Supply Chain, Security**

I. Introduction

As open-source software becomes increasingly integrated into commercial codebases, its ubiquity in software development presents significant security challenges, highlighted by the recent Open Source Security and Risk Analysis (OSSRA) report by Synopsys [1]. This report reveals that 96% of audited codebases contain open-source components, underscoring a concerning increase in high-risk vulnerabilities since 2019. Similarly, findings from the National Telecommunications and Information Administration (NTIA) [2] indicate the prevalence of outdated open-source components, compounding the difficulty of managing vulnerabilities effectively.

In response, the Software Bill of Materials (SBOM) has increasingly been recognized as an essential tool for enhancing software security and managing risks within the software supply chain [3]–[5]. An SBOM provides a detailed inventory of software components, which enhances transparency and facilitates the rapid identification and effective mitigation of vulnerabilities. Endorsed by initiatives from the Cybersecurity and Infrastructure Security Agency (CISA) and the National Institute of Standards and Technology (NIST), SBOMs are pivotal in transitioning towards a more secure and resilient software supply chain. However, despite their ability to provide visibility into software components, SBOMs lack detailed insights into the specific usage of these components within applications. This limitation, noted in previous research [6], suggests that while current SBOM security tools can detect vulnerabilities by merely comparing component versions with existing vulnerability databases (*e.g.,* OSV), they fail to confirm if an application's actual utilization of these components is ***truly*** impacted by the listed vulnerabilities in Common Vulnerabilities and Exposures (CVEs). This uncertainty can lead to inefficiencies in SBOM operations and potentially result in the misallocation of developer resources, as illustrated in Figure 1. This misallocation often results in either an overemphasis on mitigating irrelevant vulnerabilities (*e.g.,* `CVE-1` labeled as Critical Severity) or the neglect of relevant ones (*e.g.,* `CVE-2` labeled as Medium Severity) due to an incomplete understanding of the component's application-specific use. Consequently, the primary challenge is to bridge the information disparity between the dependency listings of SBOMs and the specific vulnerability details from CVEs.

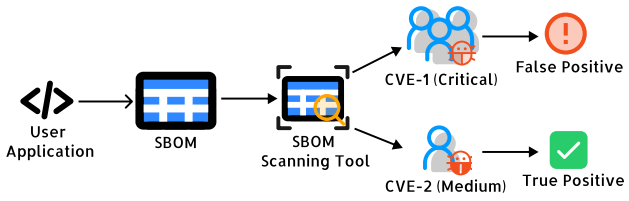

Fig. 1: Overview of resource misallocation in vulnerability investigations. *"False Positive"* indicates vulnerability (`CVE-1`) does not impact the *"User Application"*, while *"True Positive"* applies to vulnerability (`CVE-2`), which does affect it.

In this paper, we present `CovSBOM`, a novel tool designed to address the critical issue of software vulnerability management by investigating the specific code coverage of third-party dependencies within Java applications. `CovSBOM` integrates the comprehensive visibility of SBOMs with the detailed insights from CVEs, thereby enhancing the understanding and management of software vulnerabilities. Our focus on Java is driven by several compelling factors. First, it ranks among the top-three languages globally according to most recognized metrics [7]. Second, its well-established ecosystem of third-party dependencies, primarily managed through Maven or Gradle, is crucial in sectors such as government, financial services, and enterprise software systems developments. While our choice of Java highlights its complexity and the intricate nature of its ecosystem, making it a challenging yet insightful

✉Yuchen Zhang is the corresponding author

2332-6549/24/$31.00 ©2024 IEEE
DOI 10.1109/ISSRE62328.2024.00031

target, the theoretical foundation of our tool is universal and adaptable to other programming languages like Rust and Golang. This adaptability ensures that our approach can be extended to diverse programming environments, highlighting our tool's modularity, scalability, and maintainability.

Central to our framework are two essential components. First, we have developed a lightweight static analyzer to address the shortcomings of existing Java code coverage tools. While conventional tools perform well on the main application—specifically, the source code implemented by developers themselves—they fall short in analyzing code coverage for imported libraries. In contrast, our analyzer is explicitly tailored for conducting an in-depth code coverage analysis of Java third-party libraries. By providing a function as root entry, it enables developers to achieve a detailed analysis, down to the granularity of line coverage of all utilized third-party libraries. We chose static analysis to ensure a conservative approach in our evaluations, aimed at minimizing the risk of causing false negatives. Second, the outcomes of this analysis are integrated back into the SBOM, supporting both CycloneDX and SPDX formats, thereby enriching its content with precise tracking of individual function usage within third-party libraries. This enhancement significantly bridges the gap between the SBOMs' overarching component inventories and the granular vulnerability intelligence provided by CVEs. By delivering a context-driven analysis that aligns SBOMs with existing CVEs, our tool significantly reduces the incidence of irrelevant vulnerability alerts, thereby focusing patching efforts on true positive CVEs.

To assess the feasibility and utility of our tool, we conducted a comprehensive evaluation of `CovSBOM` from multiple dimensions. First, we collected 23 renowned open-source Java applications, which contain 1614 dependencies and are widely utilized in the industry. These applications have been confirmed as integral components of infrastructure and software ecosystems by our industry partners' development teams. Subsequently, we generated SBOMs for these applications and then applied the state-of-the-art SBOM analyzing tools, namely OWASP Dependency-Track [8], Grype [9], Vulert Vulnerability Scanner [10], and Bomber [11], identifying 145 potential vulnerability reports with CVEs. Second, we ran `CovSBOM` on each application to obtain code coverage of third-party libraries and integrate the results back into the *"External Reference"* field of the SBOM. Third, we correlated the CVEs with our code coverage analysis to determine if the vulnerabilities are true positives or not. Finally, `CovSBOM` identified 105 vulnerabilities that were false positives, thereby enhancing the precision of vulnerability detection by approximately 72%.

In summary, our contributions are as follows:

- We designed and implemented a static analysis tool that enhances SBOMs by integrating the application's code coverage with the dependencies or third-party libraries.
- We conducted an extensive evaluation of our tool. The results demonstrated that our tool can significantly reduce the rate of false positive alerts regarding software vulnerabilities reported by SBOM security tools.
- Our experimental datasets and the source code of our tool are made publicly available in the repository at https://github.com/Yunzez/CovSBOM.

II. Background and Motivation

A. Software Bills of Material

Beginning in May 2021, President Joe Biden's executive order has significantly encouraged the adoption of Software Bills of Materials (SBOMs) among various practitioners, including software developers, security analysts, etc. Derived from the manufacturing sector, where a Bill of Materials (BOM) [12] provides a comprehensive inventory of all sub-assemblies and components within a parent assembly, the SBOM serves as the software equivalent. It plays a crucial role in enhancing the security of the software supply chain.

There are primarily three standard formats for SBOMs: a) Software Package Data Exchange (SPDX) [13], an open-source standard managed by the Linux Foundation and primarily aimed at ensuring software license compliance, b) CycloneDX [14], introduced by OWASP in 2017, which emphasizes the security aspects of software components, and c) Software Identification (SWID) Tagging [15], focused on providing a transparent method for software and component identification. Among these, SPDX and CycloneDX formats are notably the most prevalent, while SWID Tagging, managed by the US NIST, is less commonly used.

In this paper, we focus on both CycloneDX and SPDX standards. Our decision is driven by their rapid development, consistently updated specifications, and the broad range of tooling providers that support generating SBOMs in both formats. For clarity, Figure 2 presents a simplified segment of the CycloneDX SBOM for Java application "Spark" [16]. This example highlights the three main elements of the CycloneDX standard, which are similarly adopted by SPDX:

- **Metadata Section**: provides comprehensive information regarding the SBOM generation tool and the project context.
- **Components Section**: lists the dependencies (*i.e.,* third-party libraries) found in the project.
- **Dependencies Section**: provides an overview of the relationships among all listed dependencies.

B. SBOM Security Tools

The discovery of vulnerabilities, such as the Log4j [17] in 2021, highlights the complexities and challenges involved in securing software supply chains. This incident underscores the essential role of SBOMs and emphasizes the necessity to delve into the specialized ecosystem of SBOM security tools.

In response to these challenges, leading SBOM security tools such as OWASP Dependency-Track [8], Bomber [11], Grype [9], and Vulert Vulnerability Scanner [10] are widely used in industry. Their popularity is largely due to the open-source or free-to-use nature, with each tool offering distinct functionalities that enhance the vulnerability scanning process. OWASP Dependency-Track excels in software dependency analysis, uncovering known vulnerabilities; Grype, developed by Anchore, is a robust vulnerability scanner designed for

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.x",
  "metadata": {
  "tools": {
    "components": [ {
        "group": "@cyclonedx",
        "name": "cdxgen",
        "version": "xx.x.x",
        "publisher": "OWASP Foundation"
      }
    ]
  },
  "component": {
     "group": "com.sparkjava",
     "name": "spark-core",
     "version": "2.9.4-SNAPSHOT",
     "purl": "pkg:maven/.../spark-core@
          2.9.4-SNAPSHOT?type=bundle",
     "bom-ref": "pkg:maven/.../spark-core@
          2.9.4-SNAPSHOT?type=bundle"
    }
  },
  "components": [ {
     "group": "org.eclipse.jetty",
     "name": "jetty-server",
     "version": "9.4.48.v20220622",
     "purl": "pkg:maven/jetty-server@
          9.4.48.v20220622?type=jar",
     "bom-ref": "pkg:maven/jetty-server@
          9.4.48.v20220622?type=jar"
    },
  ]
  "dependencies": [ {
     "ref": "pkg:maven/.../spark-core
          @2.9.4-SNAPSHOT?type=bundle",
     "dependsOn": [
          "pkg:maven/.../jetty-server@
              9.4.48.v20220622?type=jar"
    },
  ]
}
```


Fig. 2: Snippet of CycloneDX standard Software Bill of Materials (SBOM) in JSON format for the Java project "Spark [16]"

container images and filesystems; Bomber integrates security analysis within the software development lifecycle; and Vulert Vulnerability Scanner has extensive scanning capabilities across multiple software platforms. The core functionality of such tools is the thorough examination of SBOMs' detailed component listings, leveraging databases such as the Open Source Vulnerabilities (OSV) to identify vulnerabilities. However, this approach—primarily matching component versions with vulnerability database entries—has its own limitations. For instance, as illustrated in Figure 2, the "Spark" project uses "`jetty-server`" library, version `9.4.48.v20220622`. And if we apply this SBOM to the security tools mentioned above, all four tools raise an alarm regarding the potential impact by the vulnerability `CVE-2023-2604`. However, merely identifying a potential vulnerability does not confirm that "Spark" is actually at risk, highlighting the limitations of current SBOM

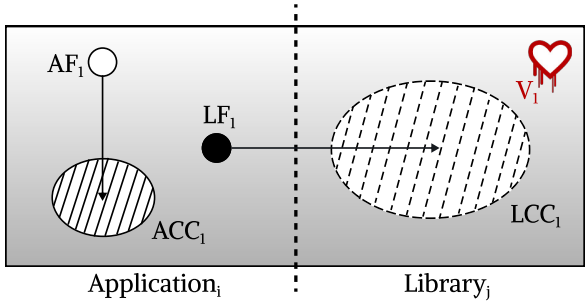

Fig. 3: Vulnerability detection for Application$_i$. `AF` denotes application function, `ACC` represents code coverage in application, `LF` indicates library function, `LCC` signifies code coverage in Library$_i$, and `V` marks the vulnerability location.

security tools in accurately assessing the real-world impact of identified vulnerabilities on projects.

C. Motivation

Driven by the uncertainty in above mentioned Java project "Spark", we conducted a detailed study through manual analysis. First, we inspected the CVE in depth and confirmed that this particular vulnerability can only be triggered by invoking function "`HttpServletRequest.getParameter()`" or "`HttpServletRequest.getParts()`" in "`jetty-server`". Second, we marked all functions defined in "`jetty-server`", but invoked by "Spark" as entry roots. We then simulated a forward slicing analysis starting from each marked function, recursively tracing the function call paths, terminating only when no further calls within a function were found, thereby generating a comprehensive call graph. We opted for manual static analysis for conservative reasons because dynamic analysis (*e.g.,* execution of pre-shipped unit tests) might not capture all code coverage due to limited test scope. In contrast, manual static analysis can start from "Spark" and assess its reachability in "`jetty-server`" on the call graph. After cross-referencing the call graph with the two vulnerable locations, we confirmed that "Spark" does not reach either of these functions, thereby indicating that, the vulnerability alert raised by SBOM security tools is a false positive.

As presented in Figure 3, current SBOM security tools, which simply map dependency utilization to CVEs, tend to signal that Application$_i$ is vulnerable due to its use of $\texttt{LF}_1$ from Library$_i$, which is known to contain vulnerability $\texttt{V}_1$. However, our comprehensive analysis reveals that the actual code coverage of $\texttt{LF}_1$ in Library$_i$ does not overlap with the identified vulnerability location $\texttt{V}_1$. This discrepancy not only highlights a critical shortfall in prevailing vulnerability management practices but also underscores the need for a more fine-grained approach that goes beyond mere dependency mapping to precisely trace how these dependencies are utilized, particularly in relation to specific function calls. The prevalent inaccuracies in SBOM analysis also pose significant operational challenges and complicate effective implementation. For instance, both Red and Blue teams in the industry could be spending considerable effort investigating false positive

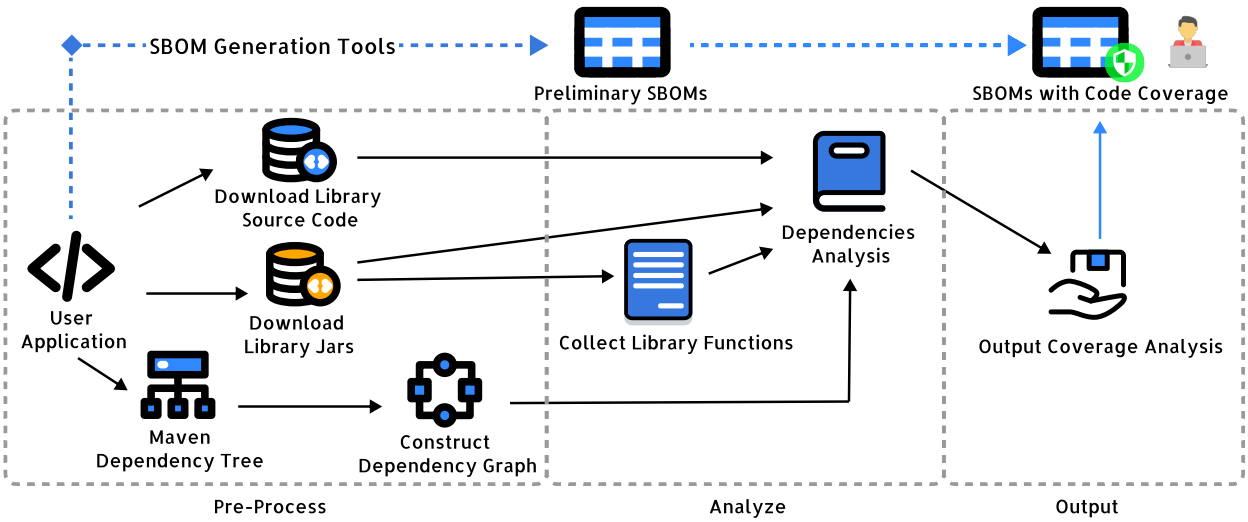

Fig. 4: Overview of the CovSBOM workflow.

TABLE I: Comparison of the leading tools for code coverage capabilities in both main applications and third-party libraries.

Tools	Coverage Analysis Capability	
	Main Apps.	Third-Party Libs.
Clover	✓	✗
Emma	✓	✗
JaCoCo	✓	✗
Cobertura	✓	✗
JCov	✓	✗
Codecov	✓	✗
Parasoft JTest	✓	✗

vulnerabilities. Without the ability to accurately determine the real impact of these vulnerabilities, organizations are forced to adopt a conservative approach, analysing every vulnerability and consequently unnecessarily allocating substantial resources. Furthermore, these investigations are rarely shared among teams within the same company, much less between different companies, which further undermines the trust in SBOMs' capability to accurately identify vulnerabilities.

Compounding these challenges is the inadequacy of current Java code coverage tools in analyzing code coverage within third-party libraries. As detailed in Table I, we conduct a survey of the prevailing Java tools for code coverage. The findings indicate that although these tools perform well in analyzing main application domain—primarily providing code coverage for unit tests—they fall short in third-party library analysis. As illustrated in Figure 3, while these tools can verify that $\texttt{AF}_1$ correlates to $\texttt{ACC}_1$, they lack the capability to establish a similar correlation between $\texttt{LF}_1$ and $\texttt{LCC}_1$, further highlighting the limitation in previous security tools' ability to provide comprehensive and accurate vulnerability assessments.

To sum up, the existing gap in the vulnerability management landscape within the SBOM ecosystem has motivated our research and underscored the necessity for a novel tool designed to bridge these disparities. Such a tool would provide a more integrated and effective method for analyzing the specific use of dependencies in relation to known vulnerabilities, thereby enhancing the ability to make informed decisions regarding vulnerability exposure and remediation strategies.

III. CovSBOM

To enhance the granularity of SBOMs and thereby reduce the rate of false positive vulnerability reports from SBOM security tools, we propose a novel tool, CovSBOM. This tool integrates code coverage information from third-party libraries directly into the SBOM while preserving the original structure and specifications to ensure compatibility with existing vulnerability scanning tools.

The workflow of CovSBOM is shown in Figure 4. Users may generate an SBOM using their preferred tools in either SPDX or CycloneDX format. Concurrently, users can utilize our tool to perform preliminary processing of the target application, which includes collecting dependencies and constructing dependency graph. Subsequently, our process involves identifying the library functions—specifically, those defined in third-party libraries. Next, all collected data is analyzed by our lightweight static analyzer to provide detailed code coverage analysis. Finally, the coverage information is integrated back into the generated SBOM. In the rest of this section, we elaborate on the designs of our tool.

A. Collect Dependencies

Since CovSBOM employs static analysis techniques to analyze dependencies, obtaining the source code for each dependency is necessary. Our analyzer, which partially utilizes JavaParser [18], a library designed to parse and navigate Java code, also requires JAR files for comprehensive analysis. To facilitate this, we use Maven commands, specifically "`mvn dependency:sources`", to download the necessary JAR files for dependencies. Once downloaded, these files are then automatically decompressed to extract the source

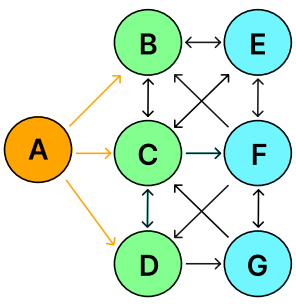

Fig. 5: Simplified dependency graph constructed by CovSBOM

code. Additionally, to align with the original SBOMs, the dependencies we collect must exactly match those listed in the SBOM. To achieve this, we follow the approach used by SBOM generation tools, which leverage another Maven command "`mvn dependency:tree`", to obtain the full list of dependencies in a tree-like structure. This command generates a dependency tree of the target application, showing relationships as depicted in Figure 5, such as (A→B→E), where A is the target application using B as a dependency, which in turn uses E. However, this tree structure does not effectively reveal subsequent layers of dependencies relationship, such as when E uses C, indicating the tree's inability to represent the full dependency relationships.

To overcome these limitations, CovSBOM constructs its own dependency graph, where each node in the graph represents a dependency. Unlike the misleadingly simple hierarchical structure implied by the term *"Dependency Tree"*, dependencies within a project exhibit a complex graph structure. This complexity arises from the interconnected nature of dependencies, where a sub-dependency may be common across multiple modules, or dependencies may be interdependent, forming a dense network. As shown in Figure 5, our graph can represent all potential call paths and further dependency relationships, such as (A→B→E→C), which a traditional dependency tree cannot accommodate due to its hierarchical nature.

Furthermore, we introduce the concepts of direct and transitive dependencies; for example, B is a direct dependency of A, while E is a transitive dependency. However, from B's perspective, E represents a direct dependency and C is a transitive dependency. Thus, the role of each dependency may change over time and depends on the specific dependency being analyzed. These concepts play a crucial role in our analysis and will be further discussed in §III-C. Additionally, it is important to note that while this study primarily utilizes Maven, the approach is equally applicable to projects managed with Gradle, due to their similar structures in dependency management. The primary distinction between them lies in the specific build commands utilized.

B. Collect Library Functions

After the preprocessing phase, we have successfully collected all the JAR files for the direct and transitive dependencies used by the target application. This setup prepares

```
1  public class ApplicationA {
2      public static void main(String[] args) {
3          DependencyB.func_x();
4          func_m();
5      }
6      public static void func_m() {
7          System.out.println("func_m from A.");
8      }
9  }
10 ------------------------------------------------
11 public class DependencyB {
12     public static void func_x() {
13         DependencyE.func_y();
14         // Call internel function func_z
15         func_z();
16     }
17     public static void func_z() {
18         System.out.println("func_z from B.");
19     }
20 }
21 ------------------------------------------------
22 public class DependencyE {
23     public static void func_y() {
24       // Call external function func_h from C
25         DependencyC.func_h();
26     }
27 }
28 ------------------------------------------------
29 public class DependencyC {
30     public static void func_h() {
31         System.out.println("func_h from C.");
32     }
33 }
```

Fig. 6: Java code snippet to present Path-2 (A→B→E→C) from Figure 5.

CovSBOM to focus on identifying library functions—those defined and implemented within third-party libraries. We collect these library functions, which serve as the entry points to direct dependencies, enabling us to further analyze transitive dependencies.

To effectively distinguish between library (*i.e.,* external) and internal functions, we first use JavaParser to convert the target program's source code into an Abstract Syntax Tree (AST) by analyzing all collected JAR files. Subsequently, we construct a HashMap where each function is mapped to its declaring type—the class or interface that originally defines the function. As we navigate through the AST, we inspect each function's declaration node to retrieve its declaring type. During this iteration, we conduct further analysis on each function's declaring type to determine its scope. Functions identified as internal—indicating that they belong to the target application—are skipped, while external functions are added to the HashMap along with their declaring types.

Consider the example shown in Figure 6, where our target application is A, and it utilizes B as a dependency. During the analysis, CovSBOM will recognize function "`func_x`" at line 3 as an external library function from the dependency B, and will add the pair (*i.e.,* function and its declaring type) to the HashMap. Conversely, when encountering function "`func_m`"

Algorithm 1: EXHAUSTIVE DEPENDENCY RESOLUTION

```
1  Initialize: LoadingBuffer = new HashMap();
2  Initialize: DoneBuffer = new HashMap();
3  Procedure ResolveDependencies(pair):
4      CalleePair = JavaParserAnalysis(pair);
5      if CalleePair exists then
6          if CalleePair.scope is external then
7              LoadingBuffer.add(CalleePair);
8              J.append(pair.declaring_type.coverage);
9              DoneBuffer.add(pair);
10             LoadingBuffer.remove(pair);
11         end
12         if CalleePair.scope is internal then
13             J.append(pair.declaring_type.coverage);
14             DoneBuffer.add(pair);
15             LoadingBuffer.remove(pair);
               /* Recursively analyze the callee pair   */
16             ResolveDependencies(CalleePair);
17         end
18     end
19 return;
20 Algorithm:
       input  : HashMap of library functions M
       output : Analysis results in JSON J
21     for Each pair in M do
22         LoadingBuffer.add(pair);
23     end
24     while LoadingBuffer not empty do
25         for each pair in LoadingBuffer do
               /* To avoid infinite loops in call path  */
26             if pair not in DoneBuffer then
27                 ResolveDependencies(pair);
28             end
29         end
30     end
31 return J;
```

at line 4, CovSBOM identifies this function as an internal function because its declaring type is found within the target application A at line 6, and therefore excludes it from further analysis. By the end of this iterative process through the AST of application A, the HashMap contains only the desired library functions, preparing it for further use in the next phase.

C. Dependency Analysis

Following the identification of library function calls within the target program, the next phase involves analyzing the code coverage of dependencies using these functions as root entry points. This step employs a lightweight static analyzer that incorporates our proposed algorithm, **Exhaustive Dependency Resolution (EDR)**, as represented in algorithm 1, which forms the core of CovSBOM. During the analysis, dependencies are categorized into two types, as introduced in §III-A: direct and transitive. Direct dependencies are those libraries with which the program directly interacts, while transitive dependencies are those that interact indirectly. This distinction is essential to our algorithm, since it utilizes recursion to analyze direct dependencies first. Due to the graph nature of dependency relationships, once direct dependencies are resolved, transitive dependencies will be treated as direct dependencies. This process will recursively continues until no further dependencies remain, ensuring a comprehensive and efficient analysis of all dependencies.

Before initiating CovSBOM's analyzer, we will first initialize two HashMaps for each node in the constructed dependency graph: LoadingBuffer to store functions pending analysis, and DoneBuffer to keep track of functions that have been analyzed, pairing each function with its declaring type (Line 1-2). The analysis process starts by iterating through the library functions HashMap anding add each function-declaring type pair into the corresponding LoadingBuffer (Line 21-23).

The analysis proceeds in a "while" loop as long as the LoadingBuffer is not empty, ensuring an exhaustive analysis (Line 24-30). Within each loop iteration, we first ensure the current pair is not already in the DoneBuffer to avoid redundant processing and prevent falling into infinite loops (Line 26). The analysis then resolves each pair's code coverage by calling ResolveDependencies (Line 27).

The resolver then utilizes the JavaParser to obtain the CalleePair—still comprising the function and its declaring type (Line 4). If the CalleePair exists, indicating potential for further analysis, we check the function's scope (*i.e,* External or Internal). If external, it is be added to the corresponding LoadingBuffer based on its declaring type. Concurrently, the code coverage information (*i.e.,* lines of code covered) for the current function is appended to the final result, and the pair is then added to the DoneBuffer, and removed from the LoadingBuffer (Lines 8-10). If the function in the CalleePair is internal, its code coverage is directly appended. The CalleePair is then moved to the DoneBuffer and removed from the LoadingBuffer. The CalleePair will be recursively analyzed by ResolveDependencies until no further CalleePair can be found (Lines 13-16).

To better illustrate the EDR algorithm and demonstrate how CovSBOM's static analyzer works, consider the example presented in Figure 6, where the call path is (A→B→E→C). Initially, CovSBOM creates two HashMaps, LoadingBuffer and DoneBuffer, for each dependency—B, E, and C. It then starts analyzing application A, classifying "func_x" at line 3 as an external function from B and adding it to the library function HashMap. Conversely, "func_m" at line 4 is identified as an internal function and is thus discarded.

The analysis then proceeds by iterating through the library function HashMap. Upon encountering "func_x", it is placed into B's LoadingBuffer. The "while" loop will be triggered due to B's LoadingBuffer not being empty, prompting the analysis of "func_x" within B. This analysis reveals "func_y" at line 13 as an external function from dependency E, which is then added to E's LoadingBuffer. Meanwhile, "func_z" at line 15 is recognized as internal function; its code coverage is analyzed and appended to the final result JSON file, along with the coverage of "func_x", which is then added to the DoneBuffer and removed from the LoadingBuffer.

As the dependency graph traversal continues, EDR observes that dependency E's LoadingBuffer is not empty, leading to the analysis of "func_y" at line 23. This function calls

```
{
  "Dependency": "groupId=org.mockito",
      "artifactId=mockito-core",
      "version=1.10.19",
      "jarPath=/mockito-core-1.10.19.jar",

  "declaringType":
        "org.mockito.ArgumentCaptor",
  "methodSignature": "getValue()",
  "methodName": "getValue",
  "lineNumber": ["153"],
  "fullExpression":
        "cookieArgumentCaptor.getValue()",
  "currentLayer": 0,
  "declarationInfo": {
  "sourceFilePath":
        "/ArgumentCaptor.java",
  "declarationStartLine": 118,
  "declarationEndLine": 120,
  "methodName": "getValue",
  "declarationSignature": "getValue()",
  "innerMethodCalls": [

  "declaringType":
        "org.mockito.CapturingMatcher",
  "methodSignature": "getLastValue()",
  "methodName": "getLastValue",
  "lineNumber": ["119"],
  "fullExpression":
        "capturingMatcher.getLastValue()",
  "currentLayer": 1,
  "declarationInfo": {
  "sourceFilePath":
        "/CapturingMatcher.java",
  "declarationStartLine": 35,
  "declarationEndLine": 42,
  "methodName": "getLastValue",
  "declarationSignature": "getLastValue()",
  "innerMethodCalls": [
        ]
    }
}
```


Fig. 7: Snippet of the coverage result produced by CovSBOM in JSON format for the Java project "Spark [16]"

"func_h" at line 25, an external function from dependency C, which is then added to C's LoadingBuffer. Concurrently, the code coverage of "func_y" is documented, "func_y" is added to E's DoneBuffer and removed from LoadingBuffer.

Following this, the analysis of "func_h" in dependency C reveals no further function calls; its code coverage is appended to the final result JSON file. Then, "func_h" is added to C's DoneBuffer and removed from LoadingBuffer. At this point, all LoadingBuffer in the dependency graph are empty, signaling the completion of the analysis. The analyzer then terminates, producing a final JSON file that encapsulates the code coverage information for functions across all dependencies. As shown in Figure 7, CovSBOM provides detailed code coverage information for each function across dependencies. This includes attributes such as "***methodName***", which specifies the function name; "***lineNumber***", the line where it was called; and "***declarationStartLine/EndLine***", the actual implementation range of that function. Additionally, "***innerMethodCalls***" is included to store information about the recursively called functions, thereby enhancing the completeness of code coverage analysis.

D. Integration of Coverage Results into SBOM

After completing the analysis, we have obtained comprehensive code coverage information for each dependency. The next step involves integrating these results back into the SBOM to enhance information granularity, detailing the exact function call chains and the specific lines of code covered by each function. To maximize the usability of CovSBOM, we offer two options tailored to different operational environments:

- **Internet Accessible Environments:** In settings where internet access is available, the code coverage results (*i.e.,* JSON files) can be uploaded to the cloud storage solutions, such as Google Cloud. Then, CovSBOM inserts a link to these results in the *"External Reference"* section of SBOMs. This approach allows for frequent updates and retrieval of data without touching the original SBOMs.
- **Air Gap Environments:** In environments lacking or forbidding internet access, CovSBOM can embed the coverage results directly into the SBOMs. Specifically, the results are incorporated into the *"External Reference"* section of the SBOMs as well, ensuring that all relevant information is contained within a single SBOM document. This allows for local verification or attestation (*e.g.,* in-toto [19]).

E. Mitigation of False Positives

The final component of CovSBOM involves leveraging code coverage information to mitigate false positive vulnerability reports. To achieve this, CovSBOM incorporates a feature that automatically scans the SBOMs after integrating the coverage results. It utilizes a HashMap where the key is the CVE ID, as reported by the security tools, and the value is the corresponding vulnerability locations. Although automating the precise identification of vulnerability locations remains challenging, we currently inspect CVEs manually. However, with advancements in Large Language Models (LLMs), this process has the potential to become automated in the future. CovSBOM then cross-references the code coverage data of each dependency against the known vulnerabilities within the corresponding HashMap to determine whether the vulnerability location falls within the code coverage range. This assessment significantly automates the detection process, determining whether reported vulnerabilities are true positives or false positives, and also facilitates easy integration into the existing scanning pipeline of SBOM security tools.

IV. Implementation and Evaluation

We have primarily implemented CovSBOM in Java, with additional components in Bash Script and Python, comprising approximately 4,000 lines of code. The tool is now publicly available at repository: https://github.com/Yunzez/CovSBOM.

TABLE II: False-Positive detection capability of CovSBOM on 145 vulnerabilities from the 23 recommended projects. The cases highlighted in *"light-gray"* indicate that these projects contain no vulnerabilities as reported by SBOM security tools.

Project	Version	CVE	FP	Project	Version	CVE	FP	Project	Version	CVE	FP	Project	Version	CVE	FP
latencyutils	2.0.4	CVE-2020-15250	✓	connect-java-sdk	2.20191120.0	CVE-2020-36518	✓	light-4j	2.1.33	CVE-2022-23302	✓	wicket	10.1.0	CVE-2020-13956	✓
time4j	5.9.4	-	-	connect-java-sdk	2.20191120.0	CVE-2022-42003	✓	light-4j	2.1.33	CVE-2022-23305	✓	wicket	10.1.0	CVE-2023-2976	✓
xchart	3.8.8	-	-	connect-java-sdk	2.20191120.0	CVE-2022-42004	✓	light-4j	2.1.33	CVE-2022-23307	✓	wicket	10.1.0	CVE-2020-8908	✓
activej	6.0	CVE-2023-33546	✓	connect-java-sdk	2.20191120.0	CVE-2020-15250	✓	light-4j	2.1.33	CVE-2023-26464	✓	wicket	10.1.0	CVE-2023-40167	✓
activej	6.0	CVE-2024-1023	☒	datafaker	2.1.1	-	-	parity	0.7.1	-	-	wicket	10.1.0	CVE-2018-18531	✓
activej	6.0	CVE-2024-1300	☒	fixture-factory	3.1.1	CVE-2020-15250	✓	password4j	1.8.1	-	-	wicket	10.1.0	CVE-2020-15250	✓
activej	6.0	CVE-2024-29025	☒	fixture-factory	3.1.1	CVE-2014-0114	✓	philadelphia	2.0.1	-	-	wicket	10.1.0	CVE-2009-2625	✓
activej	6.0	CVE-2023-6378	☒	fixture-factory	3.1.1	CVE-2019-17571	✓	simple-java-mail	8.7.1	CVE-2024-23081	✓	wicket	10.1.0	CVE-2012-0881	✓
activej	6.0	CVE-2018-14335	✓	fixture-factory	3.1.1	CVE-2021-4104	✓	simple-java-mail	8.7.1	CVE-2024-23082	✓	wicket	10.1.0	CVE-2013-4002	✓
beanmother	0.9.0	CVE-2017-18640	✓	fixture-factory	3.1.1	CVE-2022-23302	✓	spatial4j	0.9	-	-	wicket	10.1.0	CVE-2022-23437	☒
beanmother	0.9.0	CVE-2022-25857	✓	fixture-factory	3.1.1	CVE-2022-23305	✓	ta4j	0.16	CVE-2023-6481	☒	wicket	10.1.0	CVE-2017-10355	✓
beanmother	0.9.0	CVE-2022-38749	✓	fixture-factory	3.1.1	CVE-2022-23307	✓	ta4j	0.16	CVE-2024-22949	✓	xstream	1.5.0	CVE-2018-1000632	☒
beanmother	0.9.0	CVE-2023-2976	✓	fixture-factory	3.1.1	CVE-2023-26464	✓	thymeleak	3.1.3	CVE-2016-1000027	✓	xstream	1.5.0	CVE-2020-10683	☒
beanmother	0.9.0	CVE-2020-8908	✓	fixture-factory	3.1.1	CVE-2020-25638	✓	thymeleak	3.1.3	CVE-2024-22243	☒	xstream	1.5.0	CVE-2021-33813	☒
beanmother	0.9.0	CVE-2022-38751	☒	fixture-factory	3.1.1	CVE-2018-1000632	✓	thymeleak	3.1.3	CVE-2024-22233	✓	xstream	1.5.0	CVE-2022-40149	✓
beanmother	0.9.0	CVE-2022-38752	✓	fixture-factory	3.1.1	CVE-2020-10683	✓	thymeleak	3.1.3	CVE-2023-34053	✓	xstream	1.5.0	CVE-2022-40150	✓
beanmother	0.9.0	CVE-2022-41854	✓	jfairy	0.6.5	CVE-2022-42889	✓	thymeleak	3.1.3	CVE-2024-22257	✓	xstream	1.5.0	CVE-2022-45685	☒
beanmother	0.9.0	CVE-2022-1471	☒	jfairy	0.6.5	CVE-2023-2976	✓	thymeleak	3.1.3	CVE-2020-5408	✓	xstream	1.5.0	CVE-2022-45693	☒
beanmother	0.9.0	CVE-2022-38750	✓	jfairy	0.6.5	CVE-2020-8908	✓	thymeleak	3.1.3	CVE-2024-22234	☒	xstream	1.5.0	CVE-2023-1436	✓
blade	2.1.2	CVE-2024-29025	☒	jfairy	0.6.5	CVE-2023-6378	☒	thymeleak	3.1.3	CVE-2023-6378	☒	xstream	1.5.0	CVE-2022-45688	✓
blade	2.1.2	CVE-2023-34462	☒	jfairy	0.6.5	CVE-2023-6378	☒	thymeleak	3.1.3	CVE-2022-1471	✓	xstream	1.5.0	CVE-2023-5072	✓
blade	2.1.2	CVE-2018-14040	☒	jwt-java	1.2.0	CVE-2022-45688	✓	thymeleak	3.1.3	CVE-2022-25857	✓	xstream	1.5.0	CVE-2013-5816	☒
blade	2.1.2	CVE-2018-14041	☒	jwt-java	1.2.0	CVE-2023-5072	✓	thymeleak	3.1.3	CVE-2022-38749	✓	xstream	1.5.0	CVE-2019-12401	☒
blade	2.1.2	CVE-2018-14042	☒	light-4j	2.1.33	CVE-2016-6311	☒	thymeleak	3.1.3	CVE-2022-38750	✓	xstream	1.5.0	CVE-2020-25638	☒
blade	2.1.2	CVE-2019-8331	☒	light-4j	2.1.33	CVE-2023-1973	✓	thymeleak	3.1.3	CVE-2022-38751	✓	xstream	1.5.0	CVE-2019-14900	☒
blade	2.1.2	CVE-2019-11358	☒	light-4j	2.1.33	CVE-2023-5685	✓	thymeleak	3.1.3	CVE-2022-38752	✓	xstream	1.5.0	CVE-2022-41853	☒
blade	2.1.2	CVE-2020-11023	☒	light-4j	2.1.33	CVE-2023-51775	✓	thymeleak	3.1.3	CVE-2022-41854	✓	xstream	1.5.0	CVE-2020-15250	✓
blade	2.1.2	CVE-2020-13956	✓	light-4j	2.1.33	CVE-2018-14335	✓	thymeleak	3.1.3	CVE-2023-32697	✓	springdoc-openapi	2.5.1	CVE-2024-22258	☒
blade	2.1.2	CVE-2022-45688	✓	light-4j	2.1.33	CVE-2018-8088	✓	thymeleak	3.1.3	CVE-2024-29025	✓	springdoc-openapi	2.5.1	CVE-2023-52428	✓
blade	2.1.2	CVE-2023-5072	✓	light-4j	2.1.33	CVE-2018-11771	✓	thymeleak	3.1.3	CVE-2023-34062	☒	springdoc-openapi	2.5.1	CVE-2024-29025	✓
blade	2.1.2	CVE-2024-31033	☒	light-4j	2.1.33	CVE-2019-12402	✓	thymeleak	3.1.3	CVE-2023-34054	☒	springdoc-openapi	2.5.1	CVE-2018-14335	✓
blade	2.1.2	CVE-2023-2976	✓	light-4j	2.1.33	CVE-2021-35515	✓	wicket	10.1.0	CVE-2016-6345	✓	springdoc-openapi	2.5.1	CVE-2024-31033	☒
blade	2.1.2	CVE-2020-8908	✓	light-4j	2.1.33	CVE-2021-35516	✓	wicket	10.1.0	CVE-2016-6346	✓	spark	2.9.4	CVE-2023-26048	✓
connect-java-sdk	2.20191120.0	CVE-2021-28168	✓	light-4j	2.1.33	CVE-2021-35517	✓	wicket	10.1.0	CVE-2016-6347	✓	spark	2.9.4	CVE-2023-26049	✓
connect-java-sdk	2.20191120.0	CVE-2019-16942	✓	light-4j	2.1.33	CVE-2021-36090	✓	wicket	10.1.0	CVE-2017-7561	✓	spark	2.9.4	CVE-2023-40167	✓
connect-java-sdk	2.20191120.0	CVE-2019-16943	✓	light-4j	2.1.33	CVE-2024-25710	☒	wicket	10.1.0	CVE-2020-10688	✓	spark	2.9.4	CVE-2020-15250	☒
connect-java-sdk	2.20191120.0	CVE-2019-17531	✓	light-4j	2.1.33	CVE-2019-17571	✓	wicket	10.1.0	CVE-2020-1695	✓	spark	2.9.4	CVE-2020-13956	✓
connect-java-sdk	2.20191120.0	CVE-2020-25649	✓	light-4j	2.1.33	CVE-2021-4104	✓	wicket	10.1.0	CVE-2021-20289	✓	spark	2.9.4	CVE-2022-25647	✓

The rest of this section presents our evaluation of CovSBOM, centering around three primary questions:

- **Capability**: Can CovSBOM reduce the number of false positive vulnerabilities reported by SBOM security tools?
- **Scalability**: Can CovSBOM be applied to large projects while maintaining a tolerable performance overhead?
- **Usability**: Can CovSBOM provide user-friendly operation and integrate the coverage information into SBOMs flexibly?

A. Capability

To measure the detection capability of CovSBOM, we initially selected 23 open-source projects recommended by our industry partner, confirmed to simulate the actual development scenarios within their software ecosystem. For each project, we generate SBOMs in both SPDX and CycloneDX formats, then scanned these SBOMs using previously collected security tools, including OWASP Dependency-Track, Vulert Vulnerability Scanner, Grype, and Bomber. This analysis yielded a comprehensive list of potential vulnerabilities. Due to the often generic nature of these reports, specific details such as the exact location of vulnerabilities are not always available. We thus conduct a manual inspection of each reported CVE, consulting various vulnerability databases (*e.g.,* NVD, Debian Security Tracker and SUSE) to identify and document the specific functions that contain or could trigger each vulnerability. Following this preparatory step, we subjected all selected projects to CovSBOM. As detailed in Table II, out of the 145 potential vulnerabilities reported by SBOM security tools, 105 were confirmed as false positives, constituting approximately 72.4% of the total. Of the remaining 40 vulnerabilities, only 13 (about 8.97% of the total reported vulnerabilities) were verified as true positives. Additionally, 6 vulnerabilities, such as CVE-2018-14040∼14042 were unrelated to Java, pertaining instead JavaScript, and thus were beyond our analysis scope. The remaining 21 could not be definitively linked to specific buggy code locations; conservatively, we categorized these as true positives. This experiment demonstrates that the conventional approach of combining SBOMs with vulnerability scanning tools suffers from a high rate of false positive reports. However, our tool, CovSBOM, has successfully lowered the incidence of false positive reports by more than 72%, significantly enhancing the reliability of vulnerability detection and simultaneously enabling more efficient allocation of security resources across industries.

B. Scalability

CovSBOM applies static analysis to target projects; however, this analysis may be affected by the complexity and size of modern software projects that often integrate hundreds of third-party libraries. To this end, the projects we selected are sufficiently large to stress test scalability. As presented in Table III, CovSBOM can successfully analyze both small projects with a few dependencies (*e.g.,* "latencyutils") and large-scale projects with more than 200 dependencies, including "activej" (with 240 dependencies), "thymeleaf" (with 226 dependencies), and "wicket" (with 226 dependencies).

We also measure the time cost of running CovSBOM considering it a critical factor related to scalability, since extremely high run times would render the tool impractical for large programs. Furthermore, given the nature of frequently updated dependencies, the need to maintain stable codebases over time, and the frequent assignment of new CVEs, our tool may need to run frequently on continuous integration systems. As shown in Table III, CovSBOM can complete the analysis of the majority of projects within 60 seconds. Even activej, which has the largest number of dependencies, only takes about 5 minutes to analyze. After consulting with industry experts, we believe the time length is acceptable and tolerable in practice.

TABLE III: Performance of CovSBOM on the selected projects

Project	Deps. Count	Time (Sec)
latencyutils	3	5s
time4j	4	4s
xchart	27	7s
activej	240	301s
beanmother	32	8s
blade	46	25s
connect-java-sdk	27	21s
datafaker	40	19s
fixture-factory	23	11s
jfairy	37	62s
jwt-java	10	3s
light-4j	153	45s
parity	37	5s
password4j	4	4s
philadelphia	34	7s
simple-java-mail	91	6s
spatial4j	10	4s
ta4j	22	13s
thymeleak	226	60s
wicket	226	133s
xstream	90	16s
springdoc-openapi	196	33s
spark	36	17s

TABLE IV: Comparison of original and enhanced SBOM sizes

Projects	SBOM Size in KB		
	Orig. SBOM	Cov + SBOM	Increased
latencyutils	7.68	155	20.2x
time4j	8.5	26.6	3.13x
xchart	57	278	4.88x
activej	426	10862.4	25.5x
beanmother	66	215.5	3.27x
blade	98	617.5	6.30x
connect-java-sdk	61	157.2	2.58x
datafaker	86	2640.7	30.7x
fixture-factory	49	893.5	18.3x
jfairy	82	2446.8	29.8x
jwt-java	25	63.4	2.54x
light-4j	274	10156.9	37.1x
parity	70	183.5	2.62x
password4j	12	91.5	7.63x
philadelphia	57	328	5.75x
simple-java-mail	205	258	1.26x
spatial4j	25	68.7	2.75x
ta4j	45	174.8	3.88x
thymeleak	459	1979.2	4.31x
wicket	467	15260	32.7x
xstream	193	1421.8	7.37x
springdoc-openapi	410	780.9	1.90x
spark	78	712	9.13x

C. Usability

CovSBOM is designed to prioritize not only accuracy but also user convenience and integration flexibility for both developers and security teams. Usage-wise, CovSBOM offers a user-friendly command-line interface (CLI) that facilitates easy integration into existing development and security workflows. Developers can invoke CovSBOM directly from the CLI without the extensive configuration, making it accessible even to those with minimal technical expertise in SBOM management. However, CovSBOM may increase SBOM size due to its capability to embed analysis results directly into the SBOM *"external reference"* field. On average, CovSBOM can increase the original SBOM size by 11.5x, as presented in Table IV. However, the degree of increase varies, ranging from "light-4j" (approximately 37.1x) to "simple-java-mail" (approximately 1.26x). Therefore, CovSBOM also provides an alternative for scenarios where storage space is limited; developers can store results externally in centralized cloud storage services (*e.g.,* Amazon S3) and then CovSBOM embeds the links into the SBOM *"external reference"* field. This approach not only keeps the SBOM size manageable, but also ensures that detailed code coverage analyses are accessible without overburdening local storage systems. Whether operating in isolated networks or cloud-enabled environments, CovSBOM adapts flexibly, providing robust analysis capabilities without impacting system performance.

V. Related Works

A. SBOM Generation Tools

A wide range of tools are available to generate SBOMs, designed to support different programming languages. For CycloneDX format, the CycloneDX Generator [20] is recognized as the official OWASP tool. It supports an extensive array of programming languages and package managers, making it particularly suitable for multi-language projects. It offers a command-line interface (CLI) and an API server for on-demand SBOM checks, enhancing its utility in continuous integration environments. In the domain of SPDX format, tools such as the SPDX SBOM Generator [21] and the SBOM Tool [22] are endorsed by the SPDX community. These tools are also capable of managing multiple languages and package managers and are primarily designed for CLI environments. They support outputs in SPDX files, which are beneficial for compliance and licensing management.

Additionally, many tools also support both CycloneDX and SPDX formats. For examples, Syft [23] by Anchore supports a wide range of package formats and operating systems. It generates SBOMs for container images, filesystem paths, and compressed archives, making it adaptable for diverse development environments. Syft also allows SBOMs to be signed, enhancing security and integration with other tools such as vulnerability scanners. Tern [24], primarily used for analyzing container images and Docker files, perform software composition analysis to produce SBOMs detailing each layer's package information. Although flexible in output formats, its focus on containers may be limiting for broader application uses. ScanCode Toolkit [25], another open-source tool, scans codebases for licenses and copyright and is capable of generating detailed reports in both SPDX and CycloneDX formats. Its comprehensive scanning capabilities are designed to help developers ensure software compliance before distribution.

Our tool, CovSBOM, complements and enhances the existing landscape of SBOM generation tools by integrating code coverage analytical results into the SBOMs generated by any

of these tools. This compatibility ensures that both tooling providers and developers can directly incorporate CovSBOM into their existing environments with minimal effort and without any modifications to their current implementations.

B. Dependency Visualization Tools

LCM: As evidenced through collaboration with an industry partner in the financial sector, the firm is actively working to bridge the gaps among its myriad business applications, their open-source software dependencies, and the associated vulnerabilities. The internally developed tool, Life Cycle Management (LCM), plays a important role in both vulnerability management and the enhancement of lifecycle management for business applications that incorporate open-source components. Through strategic resource allocation, LCM helps maintain the integrity of business applications and minimizes the attack surface. This tool is pivotal for private sector entities that rely on open-source software to maintain competitiveness and ensure security. The firm is engaging in discussions to consider the integration of CovSBOM into its LCM product line.

GUAC: In the ecosystem of SBOMs generation and analysis, the tool **G**raph for **U**nderstanding **A**rtifact **C**omposition [26] represents a significant advancement. Developed to enhance the visibility and manageability of dependencies within software projects, GUAC utilizes a graph-based approach to trace and document the relationships between software components. GUAC's innovative utilization of graph algorithms allows it to identify indirect dependencies that are often overlooked by traditional flat-list SBOM generators. However, because it does not use static or dynamic analysis, it also suffers from the issue of false positive vulnerability reports. We are actively engaging with the GUAC community to integrate CovSBOM into their existing pipeline, aiming to reduce these inaccuracies and enhance overall vulnerability management.

VI. Conclusion

This paper first presents a case study on the false positive vulnerability reports generated by SBOM security tools, shedding light on the primary reasons for these inaccuracies. Inspired by this study, the paper introduces CovSBOM, a tool specifically designed to enhance SBOMs by integrating comprehensive code coverage analysis, thereby significantly reducing the rate of false positives. Unlike existing techniques that merely map dependency versions to CVEs, CovSBOM's lightweight static analyzer effectively analyzes the code coverage of third-party libraries. Our evaluation demonstrates CovSBOM's robust capability, showing that it can reduce around 70% of false positive vulnerability reports in large-scale Java projects, while maintaining decent scalability and usability.

VII. Acknowledgment

We thank the internal reviewers from the Depository Trust & Clearing Corporation (DTCC): Preethi Sampath, Lei Shen, Sridevi Pudhiyanayagam, and Derek Brown. We also extend our gratitude to Michael Lieberman from Kusari and our anonymous reviewers for their feedback. This research was supported by the National Science Foundation (Grant#: TI-2346219; CNS-2001161; CNS-2054692). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] B. Fred, "2024 open source security and risk analysis report." https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html, 2 2024.
[2] É. Ó. Muirí, "Framing software component transparency: Establishing a common software bill of material (sbom)," *NTIA, Nov*, vol. 12, 2019.
[3] Y. Hasan, "Realities of sbom: What is under the hood of sbom." https://apps.dtic.mil/sti/citations/trecms/AD1201272, 5 2023.
[4] L. J. Camp and V. Andalibi, "Sbom vulnerability assessment & corresponding requirements," *NTIA Response to Notice and Request for Comments on Software Bill of Materials Elements and Considerations*.
[5] J. T. Stoddard, M. A. Cutshaw, T. Williams, A. Friedman, and J. Murphy, "Software bill of materials (sbom) sharing lifecycle report," Idaho National Lab.(INL), Idaho Falls, ID (United States), Tech. Rep., 2023.
[6] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, and D. Poshyvanyk, "Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623347
[7] Stephen, Cass, "The top programming languages 2023 python and sql are on top, but old languages shouldn't be forgotten," https://spectrum.ieee.org/the-top-programming-languages-2023, 2023.
[8] The OWASP community, "Owasp dependency-track: is an intelligent component analysis platform," https://dependencytrack.org/, 2023.
[9] The Anchore, Inc., "Grype is an open source vulnerability scanner for container images and filesystems," https://github.com/anchore/grype.
[10] The Vulert Community, "Vulert vulnerability scanner," https://vulert.com/abom, 2023.
[11] S. DJ and S. Mikhail, "bomber is an application that scans SBOMs for security vulnerabilities." https://github.com/devops-kung-fu/bomber.
[12] J. Jiao, M. M. Tseng, Q. Ma, and Y. Zou, "Generic bill-of-materials-and-operations for high-variety production management," *Concurrent Engineering*, vol. 8, no. 4, pp. 297–321, 2000.
[13] Linux Foundation, "System Package Data Exchange," https://spdx.dev/.
[14] OWASP, "OWASP CycloneDX Software Bill of Materials (SBOM) Standard," https://cyclonedx.org/, 2017.
[15] NIST, "Software Identification (SWID) Tagging," https://csrc.nist.gov/projects/software-identification-swid, 2009.
[16] W. Per and H. Axel, "Spark - a tiny web framework for Java 8," https://github.com/perwendel/spark, 2023.
[17] CISA, "Apache Log4j Vulnerability Guidance." https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance, 2021.
[18] Federico, Tomassetti, "Javaparser: Tools for your java code," https://github.com/javaparser/javaparser, 2015.
[19] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1393–1410.
[20] The OWASP Community, "CycloneDX Generator," https://github.com/CycloneDX/cdxgen, 2017.
[21] P. Nirav, K. Pratik, and H. Khalifa, "SPDX Software Bill of Materials (SBOM) Generator," https://github.com/opensbom-generator/spdx-sbom-generator, 2021.
[22] Microsoft, "Sbom tool," https://github.com/microsoft/sbom-tool, 2022.
[23] G. Alex, Z. Keith, and D. Alfredo, "syft: A cli tool and go library for generating a software bill of materials (sbom)," https://github.com/anchore/syft, 2020.
[24] J. Rose and R. Michael, "Tern is a software package inspection tool that can create a sbom," https://github.com/tern-tools/tern, 2019.
[25] The nexB Community, "scancode-toolkit: A typical software project often reuses hundreds of third-party packages," https://github.com/nexB/scancode-toolkit, 2019.
[26] The Kusari Company, "Guac: Graph for understanding artifact composition," https://github.com/guacsec/guac, 2022.