JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Moving Beyond Static Inventories: AST-Based Vulnerability Reachability Analysis for Software Supply Chain Security

[1]Sujal Walanj, [2]Ankit Javeri
[1]*M.Sc. Cyber Security*, [2]*Assistant Professor*,
[1,2]Department of Advanced Computing, [1,2]Nagindas Khandwala College, [1,2]University of Mumbai, Maharashtra, India

Abstract: Modern software applications increasingly rely on third-party and open-source software components, making software supply chain security an important area of cybersecurity. Software Bill of Materials (SBOM) and dependency-based vulnerability management tools are commonly used to identify software components and match them with known vulnerabilities. However, traditional approaches primarily focus on determining whether a vulnerable component and version are present within an application. The presence of a vulnerable library does not necessarily indicate that the vulnerable functionality is actually used or reachable by the application. This research proposes an AST-Based Vulnerability Reachability Analysis approach to provide additional context for software supply chain vulnerability management. The proposed system combines dependency and SBOM information with Abstract Syntax Tree (AST)-based source code analysis. The system identifies third-party components and associated vulnerability information, analyzes Python source code to extract imported libraries, function calls, and API calls, and compares detected calls with known vulnerable APIs. Based on this comparison, vulnerabilities are classified as reachable or unreachable within the scope of the analyses source code. Reachable vulnerabilities are assigned higher priority, while vulnerabilities for which the mapped vulnerable API is not detected are assigned lower priority.

Keywords - Software Supply Chain Security, Software Bill of Materials (SBOM), Abstract Syntax Tree (AST), Vulnerability Reachability, CVE, Vulnerability Management, Static Analysis, Vulnerability Prioritization.

1. INTRODUCTION:

Modern software development increasingly depends on third-party and open-source software components. These components allow developers to reduce development time and reuse existing functionality; however, they also introduce software supply chain security risks. A vulnerability present in a third-party library can potentially affect multiple applications that depend on that component. Software Bill of Materials (SBOM) approaches provide a structured inventory of software components and dependencies used within an application. Vulnerability management tools can compare these components and their versions with known vulnerability information, such as Common Vulnerabilities and Exposures (CVEs), to identify potentially affected software. However, component-level vulnerability identification alone does not provide complete information about the practical relevance of a vulnerability to a specific application. In many cases, a vulnerable library may be present as a dependency, but the vulnerable function or Application Programming Interface (API) may not be called by the application's source code. Traditional dependency-based scanners can therefore generate findings based primarily on component presence and version matching. Security teams may need additional investigation to determine whether the vulnerable functionality is relevant to the application. This research proposes an approach that combines software component identification with Abstract Syntax Tree (AST)-based source code analysis. The system analyzes Python source code to identify imported libraries, function calls, and API calls. Vulnerability information associated with detected dependencies is then compared with the API calls identified during source code analysis. This comparison provides reachability context by determining whether a mapped vulnerable API is detected within the analyzed source code. The proposed system classifies vulnerability findings as reachable or unreachable according to the prototype's static-analysis criteria. Findings associated with detected vulnerable API calls can be prioritized for further investigation, while findings whose mapped APIs are not detected can be treated as lower-priority findings. This approach aims to move beyond static software inventories and provide additional context for software supply chain vulnerability management.

2. LITERATURE REVIEW

2.1 Existing Tools and Methodologies:

Existing software supply chain security and vulnerability management approaches primarily rely on Software Bill of Materials (SBOM), dependency analysis, Software Composition Analysis (SCA), vulnerability databases, and vulnerability scanning tools to identify third-party and open-source components used within software applications. These approaches provide visibility into software dependencies, component versions, and associated security risks by comparing identified components with known vulnerability information such as Common Vulnerabilities and Exposures (CVEs) [4], [5]. SBOM-based methodologies have become

increasingly important for improving transparency and managing software supply chain risks, while modern vulnerability scanners and dependency management tools support automated identification of potentially affected software components [3], [4]. However, most traditional approaches primarily determine whether a vulnerable component or version is present within an application and do not provide sufficient information regarding whether the vulnerable function or Application Programming Interface (API) is actually used by the application's source code [1], [2]. As a result, component-level vulnerability scanning can generate findings that require additional investigation to determine their practical relevance and priority.

2.2 Related Research Papers:

Existing research has explored SBOM generation, dependency analysis, vulnerability detection, and software supply chain security to improve visibility into third-party components [3], [4], [5]. Several studies focus on code reachability and vulnerable API analysis to determine whether vulnerable functionality is actually relevant to an application [1], [2], [6]. Research on VEX and SBOM tools further highlights the importance of contextual information and challenges related to vulnerability applicability and tool consistency [7], [8]. However, many approaches focus on these areas separately and may require complex analysis techniques. This creates a research gap for a practical system that combines dependency information with AST-based source-code analysis to identify vulnerable API usage and classify vulnerabilities based on reachability context.

3. PROBLEM STATEMENT:

Modern software applications rely heavily on third-party and open-source components. Traditional vulnerability scanners and SBOM-based approaches identify vulnerable libraries mainly by matching component names and versions with known CVEs [1], [4]. However, the presence of a vulnerable component does not always mean that the vulnerable function or API is actually used by the application [1], [2], [6]. This can result in unnecessary alerts and difficulty in prioritizing vulnerabilities. Therefore, there is a need for an approach that combines dependency information with source-code analysis to determine whether vulnerable functionality is reachable and provide more accurate vulnerability prioritization [1], [2].

4. OBJECTIVES OF THE RESEARCH:

The main objective of this research is to improve software supply chain vulnerability analysis by moving beyond traditional component and version-based vulnerability detection [1], [4]. The proposed research aims to identify third-party dependencies and their associated vulnerabilities while analyzing Python source code using Abstract Syntax Tree (AST) techniques to detect imported libraries, function calls, and API usage [1], [2], [6]. By comparing detected API calls with known vulnerable APIs, the system aims to determine whether vulnerable functionality is reachable within the analyzed application [1], [2], [6]. The research further aims to classify vulnerabilities as reachable or unreachable, support improved vulnerability prioritization, and provide the analysis results through database storage and a web-based interface.

5. PROPOSED METHODOLOGY AND SYSTEM ARCHITECTURE:

The proposed methodology combines dependency-based vulnerability detection with Abstract Syntax Tree (AST)-based source code analysis to provide reachability context for software supply chain vulnerabilities. The system first accepts the source code or project files and identifies third-party libraries and dependencies using requirements or SBOM information. The identified components are then compared with vulnerability and CVE data to detect potentially affected libraries. The source code is simultaneously analyzed using AST techniques to extract imported libraries, function calls, and API calls. The detected calls are compared with the vulnerable APIs associated with the identified vulnerabilities to determine whether the vulnerable functionality is present within the analyzed application. Based on this comparison, vulnerabilities are classified as reachable or unreachable and prioritized accordingly. The final vulnerability results are stored in a database and presented through a web-based interface, allowing developers to view the detected libraries, vulnerability information, reachability status, and vulnerability analysis results. This methodology combines component-level inventory information with source-code-level analysis to provide more contextual vulnerability prioritization [1], [2], [6].

5.1 Overall Methodology:

The overall methodology of the proposed system begins with the developer providing a software project or source code for analysis. The system identifies the third-party dependencies using requirements files or SBOM information and compares the detected components with available vulnerability and CVE data. At the same time, the Python source code is analyzed using Abstract Syntax

Tree (AST) techniques to extract imported libraries, function calls, and API calls. The detected API calls are then compared with the vulnerable APIs associated with the identified vulnerabilities to determine whether the vulnerable functionality is used within the application. Based on the analysis, each vulnerability is classified as reachable or unreachable, allowing reachable vulnerabilities to receive higher priority for further investigation. Finally, the vulnerability results are stored in the database and displayed through the web-based interface, providing developers with a clear view of dependencies, detected vulnerabilities, and their reachability status.

5.2 System Architecture:

The proposed system follows a layered architecture that integrates source code analysis, dependency identification, vulnerability detection, reachability analysis, database storage, and a web-based interface. The developer provides a software project through the web application, where the system processes the uploaded project files and identifies dependencies using requirements files or SBOM information. The vulnerability detection module compares the identified components with available vulnerability and CVE data. The AST analysis module then analyzes Python source code to extract imported libraries, function calls, and API calls. These results are passed to the vulnerability matching and reachability analysis modules, which determine whether the vulnerable API is detected within the application's source code. Based on the result, vulnerabilities are classified as reachable or unreachable and prioritized accordingly. The final scan results are stored in a SQLite database and displayed through the web interface, providing the developer with information about identified libraries, detected vulnerabilities, and their reachability status. The architecture therefore connects component-level vulnerability detection with source-code-level analysis to provide additional context for vulnerability prioritization.

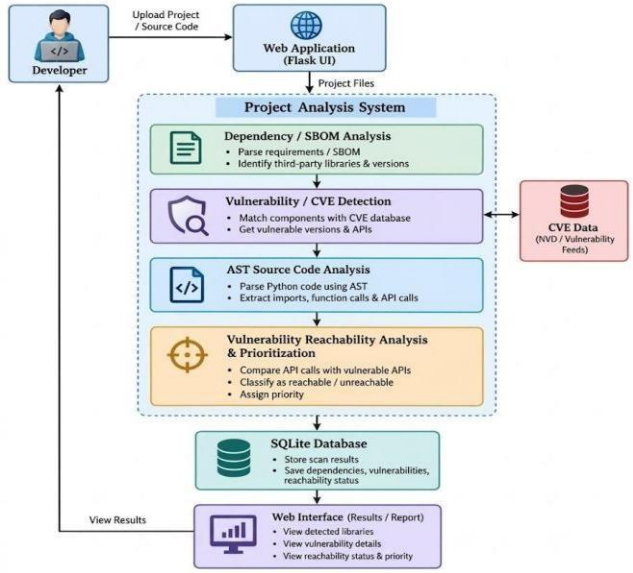

Figure 5.2: System Architecture Diagram

5.3 System Components:

Component	Description
Web Application & Project Upload	Provides a web-based interface that allows developers to upload software projects and initiate vulnerability analysis.
Dependency & SBOM Analysis	Identifies third-party libraries and software dependencies using requirements files or SBOM information.
Vulnerability & CVE Detection	Compares identified dependencies with vulnerability and CVE information to detect potentially affected components.
AST Source Code Analysis	Analyzes Python source code using Abstract Syntax Tree (AST) techniques

Component	Description
Vulnerability Reachability Analysis	Compares detected API calls with mapped vulnerable APIs to determine whether vulnerable functionality is reachable within the application.
Database & Result Management	Stores scan information and vulnerability analysis results in the SQLite database for future reference and reporting.
Results & Reporting Interface	Displays detected dependencies, vulnerabilities, reachability status, and analysis results through the web-based interface.

5.4 Flow Chart Diagram:

The workflow of the proposed system begins with the developer uploading a software project through the web application. The system then identifies project dependencies and analyzes SBOM or requirements information to detect potentially vulnerable components and associated CVEs. The Python source code is analyzed using Abstract Syntax Tree (AST) techniques to identify imported libraries, function calls, and API calls. These results are compared with mapped vulnerable APIs to determine whether the vulnerable functionality is reachable or unreachable. The vulnerabilities are then prioritized based on the reachability analysis, and the final scan results are stored in the SQLite database. Finally, the detected libraries, vulnerabilities, reachability status, and priority information are displayed to the developer through the web-based results and reporting interface.

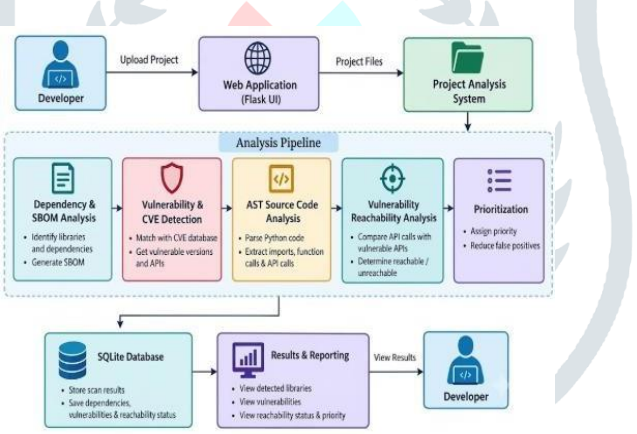

Figure 5.4: Flow Chart Diagram

5.5 Data Processing and Analysis

The proposed system processes the uploaded Python project in multiple stages. First, the system identifies dependencies from the requirements.txt file or SBOM and matches them with available vulnerability information. Next, Python source files are analyzed using the **AST module** to identify imports, function calls, and API calls. The detected API calls are compared with the vulnerable APIs associated with identified vulnerabilities.

Based on this comparison, vulnerabilities are classified as Reachable or Unreachable. The scan results are then stored in the SQLite database and displayed through the Flask web interface, allowing the developer to view the detected dependencies, vulnerabilities, and reachability status**.**

6. IMPLEMENTATION DETAILS:

The proposed AST-Based Vulnerability Reachability Analysis system was implemented using Python as the primary programming language. The system uses a modular architecture consisting of dependency analysis, SBOM parsing, vulnerability detection, AST-based source code analysis, vulnerability matching, database management, and a web-based interface. Python's built-in AST module is used to analyze source code and identify imported libraries, function calls, and API calls. Dependency information is obtained from project requirements files or SBOM data and compared with vulnerability information to identify potentially affected components. The vulnerability matching module compares detected API calls with mapped vulnerable APIs and classifies vulnerabilities based on

their reachability status. Scan results, including project information and vulnerability findings, are stored in a SQLite database. A Flask-based web application provides an interface for uploading projects, initiating scans, and viewing vulnerability analysis results. The implemented system therefore integrates component-level vulnerability detection with source-code-level analysis in a single prototype for contextual vulnerability prioritization.

6. EXPERIMENTAL EVALUATION:

The proposed system was evaluated using multiple Python test projects containing different dependency and API usage scenarios. The experiments examined whether the system could correctly identify project dependencies, detect mapped vulnerabilities, analyze source code using AST, and determine whether vulnerable APIs were reachable. Test cases included projects where a vulnerable library was present and its mapped API was called, as well as cases where the vulnerable library was present but the mapped API was not used. The experimental results demonstrated that the system could distinguish between reachable and unreachable vulnerability findings based on the detected API calls. The evaluation also verified the integration of dependency analysis, vulnerability matching, AST analysis, database storage, and the web-based interface. These experiments demonstrate the practical working of the proposed prototype and its ability to provide additional reachability context beyond traditional dependency-based vulnerability detection.

6.1 Evaluation Setup:

The proposed system was evaluated using Python-based test projects containing different combinations of third-party dependencies and API calls. The experimental environment included Python, the built-in AST module for source code analysis, Flask for the web-based interface, and SQLite for storing scan results. Test projects were analyzed using requirements or dependency information and predefined vulnerability data containing vulnerable libraries, versions, and associated APIs. The evaluationfocused on verifying whether the system could correctly identify dependencies, detect mapped vulnerabilities, extract API calls from source code, and classify vulnerabilities as reachable or unreachable based on API usage. The proposed system was evaluated using Python test projects with different dependencies and API calls. The environment included Python, AST, Flask, and SQLite. The evaluation focused on dependency identification, vulnerability detection, API extraction, and classification of vulnerabilities as **reachable or unreachable** based on API usage. The evaluation environment included **Python, the built-in AST module, Flask, and SQLite**. Test projects were provided with dependency information and predefined vulnerability data containing vulnerable libraries, versions, and APIs. The evaluation focused on checking dependency identification, vulnerability detection, API extraction, and classification of vulnerabilities as **reachable or unreachable** based on their usage in the source code.

6.2 Evaluation Metrics

The system was evaluated using the following key metrics:

• **Dependency Detection Accuracy:** Correct identification of third-party libraries and dependencies used within the analyzed software project.

• **Vulnerability Detection Accuracy:** Correct matching of identified libraries and versions with the available

vulnerability and CVE information [1], [2].

• **AST Analysis Accuracy:** Correct identification of imported libraries, function calls, and API calls from the Python source code [1], [6].

• **Vulnerability Reachability Accuracy:** Correct classification of vulnerabilities as reachable when the mapped vulnerable API is detected in the source code and unreachable when it is not detected [1], [2], [6].

• **Vulnerability Prioritization:** Ability of the system to provide additional context for prioritizing reachable

vulnerability findings over unreachable findings [1], [2].

• **Scan Result Storage:** Successful storage and retrieval of scan information and vulnerability analysis results using the SQLite database.

• **Web Interface Functionality:** Ability of the web

application to upload projects, initiate analysis, and display detected vulnerabilities and their reachability status.

6.3 Test Case:

The table below provides Test Cases : -

Test Case ID	Test Description	User Type	Input / Action	Expected Outcome
TC-01	Project Upload	Developer	Upload project	Project uploaded
TC-02	Requirements Detection	System	Read requirements.txt	Dependencies found
TC-03	Dependency Analysis	System	Analyze dependencies	Libraries identified
TC-04	SBOM Analysis	System	Read SBOM data	Components extracted
TC-05	Vulnerability Detection	System	Match vulnerability data	Vulnerabilities detected
TC-06	Import Detection	System	Analyze source code	Imports detected
TC-07	Reachable Detection	System	Vulnerable API called	Marked Reachable
TC-08	Unreachable Detection	System	API not called	Marked Unreachable
TC-09	Multiple File Scan	Developer	Upload multi-file project	Files analyzed

6.4. Result and Observations:

The experimental evaluation demonstrated that the proposed system successfully analyzed Python projects and identified third-party dependencies, imported libraries, function calls, and API calls. The vulnerability matching process detected vulnerabilities associated with identified dependencies, while AST-based analysis provided additional information about whether the mapped vulnerable API was used within the source code. Test cases where the vulnerable API was detected were classified as Reachable, whereas cases where the vulnerable library was present but the mapped API was not called were classified as Unreachable. The system also successfully analyzed multiple project files, handled duplicate findings, stored scan results in the SQLite database, and displayed the final results through the web interface. These observations show that the proposed approach can provide additional reachability context beyond traditional dependency-based vulnerability detection.

7.4 Discussion:

The experimental results show that traditional dependency-based vulnerability detection alone may provide limited information about the actual relevance of a vulnerability. The proposed system adds source-code-level context by analyzing whether a mapped vulnerable API is detected within the application. This allowed the prototype to distinguish between reachable and unreachable vulnerability findings. Reachable vulnerabilities can be given higher priority for further investigation, while unreachable findings can be treated as lower-priority based on the prototype's static-analysis results. The integration of dependency analysis, AST-based source code analysis, vulnerability matching, database storage, and a web interface also demonstrates the practical implementation of the proposed methodology. Overall, the results support the use of code reachability information as additional context for improving software supply chain vulnerability prioritization.

7.5 Summary of Evaluation:

The experimental evaluation demonstrated that the proposed system successfully performed dependency analysis, vulnerability detection, AST-based source code analysis, and vulnerability reachability classification. The system was able to distinguish between reachable and unreachable vulnerabilities based on the detection of mapped vulnerable APIs within the analyzed source code. The evaluation also confirmed successful integration of multiple file analysis, database storage, and web-based result visualization. Overall, the results demonstrate that combining dependency information with AST-based analysis can provide additional context for vulnerability prioritization beyond traditional component-based vulnerability detection.

7. SECURITY ANALYSIS:

The proposed system was designed to support secure handling of uploaded software projects and vulnerability analysis results. Project files are processed within the application environment, while scan information and vulnerability findings are stored in a SQLite database. The system focuses on identifying potentially vulnerable dependencies and analyzing source code without modifying the uploaded project files. AST-based analysis provides a static approach for examining imports, function calls, and API usage, reducing the need to execute potentially unsafe source code during analysis. The web-based interface provides controlled access to project scanning and result visualization. In addition, the reachability classification helps security teams focus attention on vulnerabilities whose mapped APIs are detected within the application source code. However, the prototype is based on static analysis and should be used as an additional vulnerability prioritization mechanism rather than as a replacement for complete vulnerability assessment and security testing.

8. CONCLUSION:

This research presented an AST-Based Vulnerability Reachability Analysis approach for improving software supply chain vulnerability management. Traditional dependency-based vulnerability detection identifies potentially vulnerable components based mainly on library and version information, but does not always determine whether vulnerable functionality is actually used within an application. The proposed system combines dependency analysis and vulnerability matching with Abstract Syntax Tree (AST)-based source code analysis to identify imported libraries, function calls, and API calls. By comparing detected API usage with mapped vulnerable APIs, the system classifies vulnerability findings as reachable or unreachable. The experimental evaluation demonstrated the successful integration of dependency analysis, vulnerability detection, AST analysis, reachability classification, database storage, and web-based result visualization. The proposed approach provides additional context for vulnerability prioritization and can help reduce unnecessary investigation of vulnerabilities whose mapped functionality is not detected in the analyzed source code. Overall, the research demonstrates the practical value of moving beyond static software inventories toward more contextual vulnerability analysis in software supply chain security.

9. FUTURE WORK:

Future work can extend the proposed system by supporting additional programming languages and software ecosystems beyond Python. The vulnerability database can be integrated with real-time sources such as the National Vulnerability Database (NVD) to provide updated CVE information automatically. The reachability analysis can also be improved by using advanced techniques such as control-flow analysis, call graph analysis, and data-flow analysis to provide more accurate results. Future versions may support larger software projects, additional SBOM formats, and more vulnerability intelligence sources. The system can also be enhanced with stronger authentication, role-based access control, automated report generation, and cloud-based deployment to support practical use in enterprise software supply chain security environments

11. REFERENCES:

[1] H. Patel, A. Snit, and M. Polychronakis, “Supply Chain Reaction: Enhancing the Precision of Vulnerability Triage Using Code Reachability Information,” 2024. https://doi.org/10.1371/journal.pone.0344098.

[2] L. Zhou, M. Dacier, and C. Konstantinou, “Reality Check on SBOM Vulnerability Management,” 2024. https://doi.org/10.1371/journal.pone.0335128.

[3] P. R. Seknametla, “Secure Supply Chain Management in DevOps: Software Bill of Materials Risks,” *International Journal of Engineering Research and Emerging Technologies*, https://doi.org/10.63282/3050-922X.IJERET-V6I2P115.

[4] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, “An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead,” https://arxiv.org/abs/2301.05362.

[5] S. E. Ponta, H. Plate, and A. Sabetta, “Detection, Assessment and Mitigation of Vulnerabilities in Open Source Dependencies,” https://doi.org/10.1016/j.jss.2024.112507.

[6] F. Zhang, L. Fan, S. Chen, M. Cai, S. Xu, and L. Zhao, "Does the Vulnerability Threaten Our Projects? Automated Vulnerable API Detection for Third-Party Libraries,". https://arxiv.org/abs/2503.15021.

[7] Y. Churakova and M. Ekstedt, "Vexed by VEX Tools: Consistency Evaluation of Container Vulnerability Scanners," https://doi.org/10.5281/zenodo.14233414

[8] M. Mirakhorli, D. Garcia, S. Dillon, K. Laporte, M. Morrison, H. Lu, V. Koscinski, and C. Enoch, "A Landscape Study of Open Source and Proprietary Tools for Software Bill. https://mehdimirakhorli.github.io/files/SBOM_Landscape.p df.

[9] L. Soeiro, T. Robert, and S. Zacchiroli, "Wild SBOMs: A Large-scale Dataset of Software Bills of Materials from Public Code," https://arxiv.org/abs/2601.05622.

[10] C. Wang, J. Wu, H. Lyu, X. Ling, T. Luo, Y. Wu, and C. Zhao, "A Large Scale Empirical Analysis on the Adherence Gap between Standards and Tools in SBOM," https://arxiv.org/abs/2507.03633.

[11] H. Cai, Q. Xiong, and S. Lian, "Research on Network Security Vulnerability Risk Contagion in Software Supply Chain Based on System Dynamics," https://doi.org/10.1007/s10664-020-09830-x.

[12] "Understanding Security Challenges in the Software Supply Chain Through Causal Relationships," https://arxiv.org/abs/2409.02753.

[13] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, Assessment and Mitigation of Vulnerabilities in Open SourceDependencies, "https://arxiv.org/abs/2503.14388.

[14] F. Zhang et al., "VAScanner: Automated Vulnerable API Detection for Third-Part Libraries," https://doi.org/10.1145/3658644.3670351.

[15] D. Cotroneo, L. De Simone, and R. Natella, "Timing Covert Channel Analysis of the VxWorks MILS Embedded Hypervisor Under the Common Criteria SecurityCertification," https://doi.org/10.1016/j.cose.2021.102307.