

Article

Software Supply Chain Risk Precise Detection Method (SSCRPDM) Based on Dynamic Bytecode Instrumentation

Rui Guo ^{1,2} , Najinsha Hu ¹, Yizhi Ma ³, Zihan Huang ¹, Fengwei Peng ¹, Gang Li ^{1,*} and Guangjun Wen ^{1,*} 

¹ School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China; 202212012404@std.uestc.edu.cn (R.G.); 202322011510@std.uestc.edu.cn (N.H.); hzh_official@163.com (Z.H.); 202212012309@std.uestc.edu.cn (F.P.)

² China Mobile Chengdu Institute of Research and Development, Chengdu 610213, China

³ School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden; yizhim@kth.se

* Correspondence: ligangpm@uestc.edu.cn (G.L.); wgj@uestc.edu.cn (G.W.)

Abstract

With the increasing softwarization of satellite payloads and the evolution of 6G Non-Terrestrial Networks (NTNs), securing the onboard software supply chain has become critical for mission-critical space communications. Traditional static software composition analysis (SCA) often generates excessive false positives in resource-constrained satellite environments. To address this issue, this paper proposes SSCRPDm, a CVE-oriented static-dynamic risk detection method based on bytecode instrumentation. SSCRPDm parses CVE semantics to guide function-level minimal instrumentation, quantifies risk by jointly considering runtime reachability, parameter controllability, and sanitizer effectiveness, and employs deterministic proof-of-concept (PoC) verification as the primary decision mechanism, with a large language model used only for auxiliary interpretation. Specifically, SCA first identifies components containing known vulnerabilities. Runtime instrumentation then traces critical invocation paths and verifies the reachability of vulnerable functions. Finally, CVE-specific PoC payloads are executed under the current defense context to determine actual exploitability. Experimental results show that SSCRPDm effectively filters alerts caused by “zombie components” and significantly improves evaluation accuracy through exploitability verification. By bridging static version matching and dynamic threat validation, SSCRPDm substantially reduces false positives and provides a precise and proactive solution for satellite software supply chain governance.

Keywords: software supply chain security; software composition analysis; bytecode instrumentation; reachability analysis; vulnerability verification



Academic Editor: Lamei Zhang

Received: 31 March 2026

Revised: 22 July 2026

Accepted: 25 July 2026

Published: 29 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction and Background

With the rapid evolution of Software-Defined Satellites (SDS) and the integration of 6G Non-Terrestrial Networks (NTN), the construction of modern space communication systems increasingly relies on a complex ecosystem of third-party and open-source components. This shift towards a component-based paradigm enables the rapid deployment of space-air-ground-sea (SAGS) connectivity services [1]. However, this architectural evolution introduces unprecedented security risks to the space software supply chain. If a core component within an orbital asset contains a vulnerability exploited by an attacker, it could lead to catastrophic consequences, such as unauthorized command execution, scientific data leakage, or total loss of satellite control [2].

In the context of autonomous space missions, traditional Static Software Composition Analysis (SCA) faces critical limitations. Current industrial solutions primarily rely on coarse-grained version matching against vulnerability databases (e.g., NVD and CVE) [3]. While effective in terrestrial environments, this approach generates an excessive number of false positives that are intolerable for resource-constrained satellite payloads [4]. Relying on static matching overlooks the actual runtime invocation context in space missions [5]. For autonomous satellites governed by Agentic AI, excessive false alerts can lead to “decision fatigue” or trigger unnecessary and risky in-orbit maintenance procedures [6]. Moreover, the traditional “patch-all” strategy incurs significant telemetry and command (TT&C) bandwidth consumption and may introduce regression risks that are impossible to remediate through physical access [7]. To address these challenges, this paper proposes a Software Supply Chain Risk Precise Detection Method (SSCRPDM) based on dynamic bytecode instrumentation, specifically optimized for intelligent space communication nodes. By implanting lightweight probes during the satellite application’s runtime, the method captures component-level call chains to accurately verify the on-orbit reachability of high-risk functions. Validated within a simulated satellite edge computing environment containing critical vulnerabilities (e.g., Log4j2, Fastjson), SSCRPDMDemonstrates its ability to precisely track risky paths and eliminate “zombie components” that are inactive in specific mission profiles. Crucially, the proposed method offers a high-confidence perception layer for Agentic AI-driven security governance, enabling autonomous systems to prioritize repairs for truly exploitable risks without human intervention, thereby ensuring the resilience and continuity of next-generation space communication networks.

Currently, the mainstream industrial solution relies on Static Software Composition Analysis (SCA) technology. This approach locates risky components by matching project dependencies against known vulnerability databases (such as NVD and CVE) [3]. While effective at rapidly identifying vulnerable components at a macro level, this method often struggles to accurately assess the exploitability of vulnerabilities. Detection logic that relies solely on coarse-grained version matching overlooks the actual invocation context of the business code [4], resulting in an excessively high false positive rate. Specifically, many components containing vulnerable code are never loaded or invoked during actual business execution [5]. This coarse-grained strategy compels development teams to upgrade or replace numerous inactive components [6], incurring significant maintenance costs and potentially introducing system regression risks due to version incompatibility [7].

In light of these challenges, this paper proposes a software supply chain risk precise detection method (SSCRPDM) based on dynamic bytecode instrumentation. This method implants probes during the application’s runtime to collect and analyze component method call chains, accurately identifying the runtime reachability of high-risk vulnerability functions in the production environment. By constructing a test environment containing typical high-risk vulnerability components such as Log4j2 and Fastjson, it has been proven that SSCRPDMDnot only achieves precise tracking of risky calls but also effectively eliminates inactive zombie components, significantly reducing the false positive rate. At the same time, this method has good scalability and low invasiveness, and can shift the focus of security repair work from all components with vulnerabilities to the truly high-risk components without changing the original business logic.

Although SCA and runtime instrumentation have been widely adopted, existing methods generally either stop at component-level version matching or use instrumentation as an isolated post-processing step, leaving static vulnerability discovery disconnected from runtime threat verification. Unlike conventional SCA tools, SSCRPDMDparses CVE semantics to convert component-level vulnerability descriptions into function-level instrumentation targets. Unlike existing IAST solutions that mainly determine path reachability,

it jointly evaluates vulnerability severity, runtime reachability, parameter controllability, sanitizer effectiveness, and exploitability. Moreover, unlike purely LLM-based analysis, SSCRPDM adopts deterministic PoC verification as the primary decision basis and uses the LLM only for rule-constrained semantic interpretation. These mechanisms establish a closed-loop process from static candidate discovery to runtime reachability analysis and exploitability confirmation.

In contrast to existing solutions that simply chain together off-the-shelf SCA and instrumentation, the major contributions and novelty of this work are summarized as follows:

- We introduce a collaborative detection architecture for on-orbit software supply chains that integrates static composition analysis, runtime bytecode instrumentation, and exploitability verification, thereby bridging vulnerable-component discovery and runtime threat confirmation in resource-constrained satellite payloads;
- We design a CVE-guided vulnerable-function extraction and risk-scoring mechanism that converts unstructured CVE/NVD descriptions into structured instrumentation targets and jointly evaluates vulnerability severity, runtime reachability, parameter controllability, sanitizer effectiveness, and exploitability for fine-grained risk prioritization;
- We propose a deterministic PoC-verification-first scheme that replays CVE-specific payloads along instrumented call chains and uses rule-constrained LLM-assisted semantic interpretation to distinguish exploitable flows from inactive or sanitized paths, reduce false positives, and support risk-oriented security maintenance for intelligent space communication nodes.

The remainder of this paper is organized as follows. Section 2 reviews related works on software composition analysis and runtime security detection. In Section 3, the motivation, problem statement, and risk-scoring model are presented. Section 4 details the proposed SSCRPDM scheme, including dependency graph construction, static vulnerability identification, runtime reachability analysis, exploitability verification, and LLM-assisted semantic assessment. Experimental results are reported in Section 5. Section 6 discusses the prerequisites, comparisons with existing approaches, and limitations. Finally, the concluding remarks are provided in Section 7.

2. Related Works

With the rapid development of information technology and the prevalence of ubiquitous applications, the scale of software systems continues to expand. Enterprises and organizations in core business sectors—such as the internet, finance, and e-commerce—are increasingly dependent on digital systems. Under the agile development paradigm, Web application development has shifted from traditional full-stack in-house development to a modular approach. Development teams universally utilize integrated open-source components, third-party libraries, and commercial Software Development Kits (SDKs) to rapidly build technical architectures [7]. According to the “2025 China Software Supply Chain Security Analysis Report” released by Qi An Xin [8], 10,320 new vulnerabilities related to open-source software were added to public vulnerability databases in 2024. In the mainstream open-source package ecosystem, the number of inactive open-source software projects reached 7,453,176, accounting for 74.5%, a trend that shows an increasing number of inactive projects. However, the overall risk remains high. These statistics indicate that the number of vulnerabilities in open-source software continues to rise, and the proportion of open-source components in the software ecosystem is constantly increasing. Attackers frequently exploit these vulnerabilities to infiltrate systems.

Currently, the mainstream detection method involves using SCA tools to analyze the composition of Web applications and identify the sources of vulnerabilities for further

risk management. SCA [9,10] is a technique that analyzes source code, build files, or dependency configuration files to extract project dependencies, including third-party components, component versions, and licensing information. It generates a Software Bill of Materials (SBOM) and uses this SBOM to determine whether the project introduces components associated with known vulnerabilities. This enables development teams to rapidly track and analyze open-source components and evaluate the software's security, reliability, and maintainability. However, CVE-based static matching itself has inherent limitations. Since the matching process mainly depends on component names, versions, and vulnerability database records, it may report vulnerable components that are never loaded or invoked at runtime, resulting in false positives. At the same time, it may also miss vulnerabilities caused by incomplete dependency declarations, inaccurate version identification, delayed CVE disclosure, or customized component modifications, leading to false negatives. These limitations directly affect the interpretation of experimental results: the initial SCA results should be regarded as candidate risks rather than confirmed exploitable threats. Therefore, additional runtime reachability analysis and exploitability verification are necessary to further refine the risk assessment.

Beyond SCA, existing Web application security detection methods are commonly divided into SAST, DAST, and IAST, which analyze vulnerabilities from static, dynamic, and interactive perspectives, respectively [11].

SAST [12,13], also known as white-box testing, can be initiated at the beginning of the application development cycle. SAST is capable of identifying complex vulnerabilities in the early development stage [14] by analyzing issues within the source code, allowing developers to fix them quickly. However, SAST usually lacks runtime context and therefore suffers from a relatively high false-positive rate.

DAST [12] is a black-box testing method. Unlike traditional static testing methods, which analyze source code or binary files, DAST evaluates the application while it is running. DAST can identify runtime vulnerabilities and usually produces fewer false positives than SAST [15]. However, DAST cannot provide developers with precise feedback regarding the vulnerable dependency, risk function, or specific code location.

IAST [12] is a gray-box scanning tool that provides higher precision in software analysis. IAST can locate vulnerability information at a finer granularity [16] and can also assist in detecting supply chain vulnerabilities during runtime. As a security detection technology combining the advantages of SAST and DAST, it offers higher precision in vulnerability detection. Nevertheless, existing IAST methods still rely heavily on whether risky execution paths are covered during testing, and they usually lack explicit CVE-oriented exploitability verification. Therefore, this paper further introduces dynamic bytecode instrumentation and PoC verification on the basis of static CVE matching, so that the impact of false positives and false negatives in the initial SCA stage can be reduced in subsequent analysis.

The comparison of the detection capabilities of SAST, DAST, and IAST is shown in Table 1. The analysis above indicates that existing methods and systems still exhibit limitations regarding real-time defense and precise monitoring. Therefore, this paper investigates a method that combines static and dynamic detection techniques. By utilizing dynamic instrumentation at runtime to achieve dynamic monitoring of risky execution paths, further ensuring the precision and traceability of component risks.

Table 1. Comparison and Analysis of Safety Detection Technology.

Comparison	SAST	DAST	IAST
detection object	source code	running application	running application
false positive rate	high	low	extremely low
coverage	high	low	medium
Intrusiveness	low	high	low
Usage cost	high	low	low
Detection rate	high	medium	relatively high

3. Motivation

3.1. Problem Statement

The Software-Defined Satellite Architecture: In this architecture, an on-board application is no longer a closed, isolated firmware image, but rather a dynamic aggregate of mission-specific code, open-source communication protocols, and third-party satellite IoT libraries. As illustrated in Figure 1, the application lifecycle—spanning design, construction, and delivery—involves collaboration across multiple departments and complex toolchains [17]. In this context, an application is no longer a closed, isolated collection of code, but rather an aggregate of in-house code, open-source components, and third-party libraries [18]. While these elements collectively constitute the software supply chain, malicious activities can infiltrate at any stage, thereby threatening the security of the entire supply chain [19].

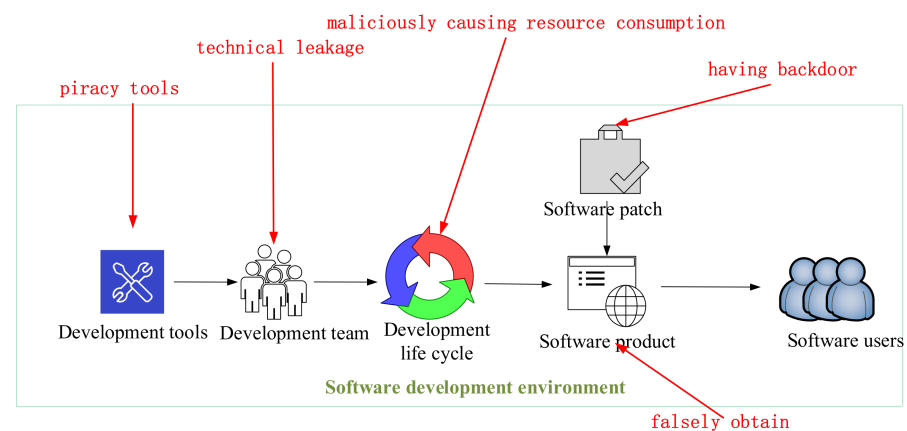
**Figure 1.** The scope of influence of the development process.

Figure 2 illustrates the application software supply chain construction process, highlighting the intricate nature of the software supply chain network where an oversight at any stage can serve as an entry point for attackers. This high degree of dependency and interconnectivity makes maintaining the security of the development environment extremely challenging. Once a foundational component in the upstream supply chain is implanted with malicious code or exposes a critical vulnerability [2], the risk propagates rapidly downstream along the dependency chain. This can lead to severe consequences, such as privilege escalation, service disruption, or the leakage of sensitive data [1].

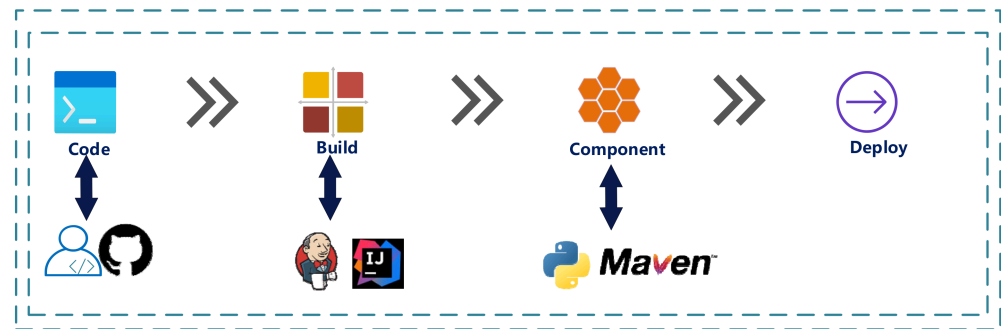


Figure 2. Software supply chain construction process.

3.2. Proposed Solution

To identify the version information of components in the supply chain. Currently, the industry primarily relies on SCA technology to address supply chain risks. These technologies identify risks by scanning project dependency files and matching component versions against CVE vulnerability databases. Although SCA excels at rapidly discovering known vulnerabilities, its limitations have become increasingly apparent when dealing with complex modern applications, primarily across the following three dimensions.

3.2.1. Lack of Runtime Reachability

In real production environments, the presence of a vulnerable component does not imply that the vulnerable function will inevitably be triggered [20]. Due to a lack of awareness regarding runtime invocation paths, existing technologies cannot distinguish between potential risks and substantive threats [21], leading to numerous uninvoked components being flagged as high-risk. This over-assessment not only interferes with security decision-making but also causes severe “alert fatigue.” Define a function to describe the real conditions under which a vulnerability is triggered, thereby contrasting the limitations of traditional SCA. For instance, define an actual security risk index for a component, which not only depends on the existence of the vulnerability but also on the reachability of its call path in space communication tasks:

3.2.2. Complex Dependency Chains

Software supply chain risks exhibit significant chain-propagation characteristics [22]. Static analysis struggles to accurately model cross-component joint-invocation relationships, leading to missed complex vulnerabilities triggered by multi-component interactions and weakening the depth and comprehensiveness of risk assessment.

3.2.3. Surging Operational Costs

Remediation recommendations based on static analysis typically involve direct upgrades or replacements [23]. However, component upgrades often involve API changes or compatibility issues, necessitating extensive regression testing of business logic [24,25]. The blind remediation strategy results in a massive waste of human resources and operational costs [26,27].

3.3. Model Building

Figure 3 presents the scene structure diagram of the software supply chain of the space communication satellite payload, clearly showing the interrelationships among the payload, software components, vulnerabilities, and attack links. In view of the dual dilemma faced by static analysis regarding accuracy and remediation costs, this paper proposes SSCRPDM based on bytecode instrumentation technology.

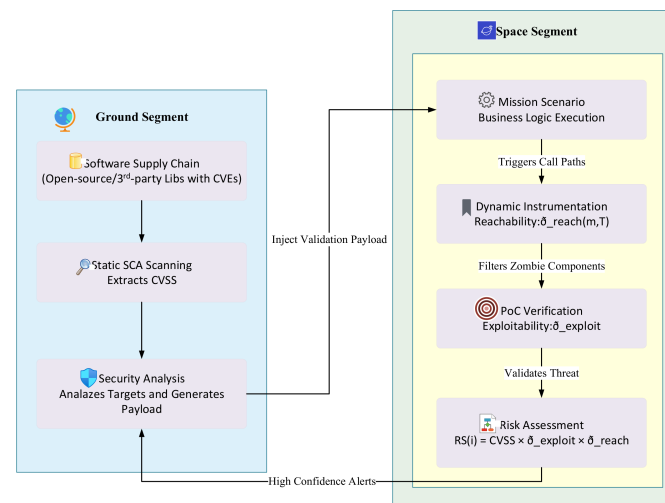


Figure 3. Software supply chain scenario for a space communication satellite payload.

The method utilizes bytecode instrumentation to monitor the actual invocation paths of components in real time, precisely identifying whether vulnerable interfaces are actually triggered, thereby eliminating invalid false positives. Furthermore, combined with an automated Proof of Concept (PoC) verification mechanism, it definitively confirms the exploitability of risks within the current security context. The study aims to focus remediation efforts on high-risk components that are genuinely invoked without altering business logic, thereby significantly reducing the security governance costs of the software supply chain while enhancing the precision of risk assessment.

To operationalize this conceptual model within the proposed SSCRPDM framework, we map these variables to specific runtime metrics. First, to eliminate “zombie components” and calculate the practical invocation probability, we define the dynamic reachability function δ_{reach} as a binary variable:

$$\delta_{reach}(m, T) = \begin{cases} 1, & \exists t \in T \text{ s.t. method } m \text{ is invoked} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Here, m represents the core risk function within component, and T denotes the satellite’s operational period or the observation time window for a specific task. Only when $\delta_{reach} = 1$ does this component be regarded as having runtime risk, thereby eliminating the useless “zombie components”.

Furthermore, reachability is not equivalent to exploitability. To quantify the actual hazard under the satellite-ground link’s defense context, we introduce $\delta_{exploit}$ as the coefficient of vulnerability exploitation effectiveness. This is determined through automated Proof of Concept (PoC) verification:

$$\delta_{exploit} = \begin{cases} 1.5, & \text{high risk} \\ 1.0, & \text{medium risk} \\ 0.5, & \text{low risk} \end{cases} \quad (2)$$

Finally, in order to accurately assess the threat that the vulnerability poses to the satellite-ground link, the final risk score $RS(i)$ is defined:

$$RS(i) = CVSS \cdot \delta_{exploit}(i) \cdot \delta_{reach}(i, t) \quad (3)$$

CVSS: The industry standard base score for this vulnerability.

The three formulas above constitute a progressive risk quantification system, whose logical correctness can be formally derived as follows:

(1) Static candidate set and runtime reachability filtering

Formula (1) defines the binary reachability coefficient δ_{reach} . From the perspective of set theory, the risky component set after reachability filtering R_1 satisfies:

$$R_1 = \{i \in R_0 \mid \delta_{reach}(m_i, T) = 1\} \quad (4)$$

where R_0 is the candidate risky component set output by static SCA. Since components with $\delta_{reach} = 0$ are never invoked during the mission period, they cannot pose any actual threat to the system. This step strictly proves mathematically that removing such components from the risk list will not introduce false negatives, which is the theoretical basis for reducing false positives in the first stage.

(2) Exploitability calibration based on PoC verification

Formula (2) introduces the exploitability coefficient $\delta_{exploit}$ to distinguish the hazard degree of reachable vulnerabilities. Even if a vulnerability is reachable ($\delta_{reach} = 1$), its actual risk varies significantly depending on whether the input is controllable and whether there is an effective sanitizer. $\delta_{exploit}$ quantifies this difference through three-level coefficients, realizing the transition from “reachable or not” binary judgment to “how dangerous it is” continuous measurement.

(3) Multi-dimensional risk fusion and prioritization

Formula (3) combines the static vulnerability severity with the two runtime correction factors

$$RS(i) = CVSS_i \delta_{reach}(m_i, T) \cdot \delta_{exploit}(i) \quad (5)$$

Zero-risk property: If $\delta_{reach}(m_i, T) = 0$, then $RS(i) = 0$. Thus, a statically matched vulnerability that is not observed at runtime is not treated as an active threat within the current operational window.

Monotonicity: For a fixed $CVSS_i$ and $\delta_{reach}(m_i, T) = 1$, $RS(i)$ increases with $\delta_{exploit}(i)$. Therefore, candidates with stronger PoC evidence receive higher risk priorities.

This set of formal models mathematically establishes the rationale for a detection pipeline consisting of static screening, dynamic reachability filtering, and exploitability calibration, thereby providing a theoretical foundation for the subsequent module design.

4. SSCRPDMM

4.1. Overview

To reduce the false-positive rate of traditional Static Software Composition Analysis (SCA) in software supply chain risk detection, this paper proposes a Software Supply Chain Risk Precise Detection Method based on Dynamic Bytecode Instrumentation (SSCRPDMM). Unlike conventional methods that only match component names and versions against CVE records, SSCRPDMM extracts risk functions from CVE descriptions, CWE categories, security advisories, patch differences, and known exploitation chains. The extracted risk functions are then used to generate a minimal instrumentation policy, capture runtime invocation paths, trace external input propagation, observe sanitizer execution, and combine PoC verification with rule-bounded LLM-assisted analysis for vulnerability risk classification [28].

Let the target application be denoted as (A), and its dependency set be defined as:

$$D = \{d_i = (c_i, v_i, p_i, r_i) \mid i = 1, 2, \dots, n\} \quad (6)$$

where c_i denotes the component name, v_i denotes the component version, p_i denotes the parent node in the dependency graph, and r_i denotes whether the dependency is direct or transitive. The CVE records in the vulnerability database are represented as:

$$V = \{vul_j = (id_j, cpe_j, V_j^{aff}, cvss_j, desc_j) \mid j = 1, 2, \dots, m\} \quad (7)$$

where id_j denotes the CVE identifier, cpe_j denotes the affected product identifier, V_j^{aff} denotes the vulnerable version range, $cvss_j$ denotes the CVSS base score, and $desc_j$ denotes the vulnerability description.

The overall workflow of SSRPDM can be formalized as:

$$L = \mathcal{C}(R_0, \Phi, I, \Omega, P), R_0 = \mathcal{S}(D, V), \Phi = \mathcal{G}(R_0), I = \mathcal{I}(\Phi) \quad (8)$$

where R_0 denotes the set of candidate risky components obtained by static matching, Φ denotes the extracted risk function set, I denotes the minimal instrumentation policy generated according to Φ , Ω denotes the runtime observations collected through dynamic bytecode instrumentation, P denotes the PoC verification result, and L denotes the final risk level.

SSCRPDM follows a six-layer workflow. First, the application dependency set D and vulnerability library V are matched to generate the candidate risk set R_0 . Second, CVE information in R_0 is parsed to extract the structured risk function set Φ . Third, Φ is used to generate a minimal instrumentation policy I . Fourth, the policy is deployed in the JVM to collect runtime records Ω and evaluate function reachability and parameter controllability. Fifth, reachable risk paths are verified through PoC replay to obtain exploitability and sanitizer-effectiveness results P . Finally, R_0 , Φ , Ω , and P are fused to calculate the risk score RS_j and determine the final risk level L . This workflow suppresses unreachable or effectively sanitized vulnerabilities while assigning higher scores to risks that are reachable, controllable, and practically exploitable.

The novelty of SSRPDM lies in three aspects. First, it extracts CVE-oriented risk functions and transforms vulnerability information from component-level matching into function-level localization. Second, it uses the extracted risk functions to guide dynamic instrumentation, making the transfer from static analysis to runtime analysis explicit and traceable. Third, it integrates PoC verification results into the structured LLM input, so that the final classification is based on static vulnerability information, runtime invocation behavior, and exploitability verification rather than isolated tool outputs. The SSRPDM structure is shown in Figure 4.

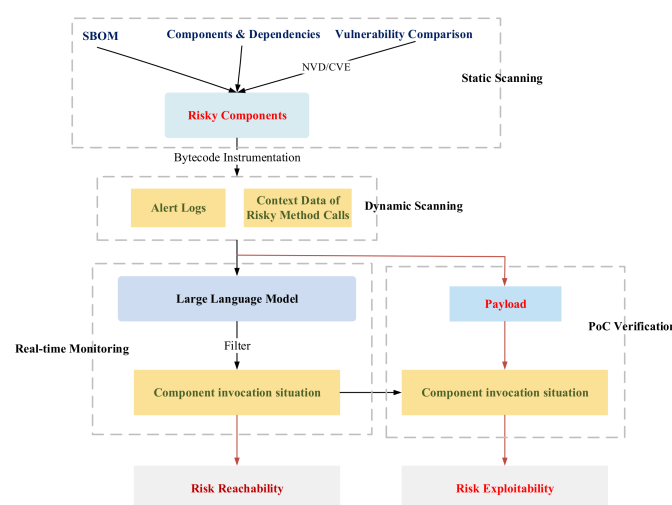


Figure 4. A SSRPDM framework for risky-component identification, runtime reachability analysis, and exploitability verification in satellite payload software.

4.2. Dependency Graph Construction

In the dependency analysis stage, SCA tools are employed to automatically parse the build files and dependency configuration files of the target application [9], as shown in Figure 5. By using the native dependency resolution capability of SCA tools, third-party components, version information, and hierarchical dependency relationships can be extracted without additional custom logic. To support subsequent vulnerability matching and risk assessment, the extracted dependency information is further represented as a directed graph:

$$G = (N, E) \quad (9)$$

where N represents the application component, third-party components, and their versioned instances, while E represents dependency relationships. If component c_a depends on component c_b , there is a directed edge:

$$e_{ab} = (c_a, c_b, type, scope) \quad (10)$$

where $type$ denotes whether the dependency is direct or transitive, and $scope$ denotes the dependency scope, such as compile, runtime, or test. For each dependency $d_i = (c_i, v_i, p_i, r_i)$, if its name matches the CPE or alias of a CVE record and its version v_i falls into the affected version set V_j^{aff} , it is added to the candidate risky component set:

$$R_0 = \left\{ (d_i, vul_j) \mid Match(c_i, cpe_j) = 1 \wedge v_i \in V_j^{aff} \right\} \quad (11)$$

where $Match(c_i, cpe_j)$ denotes a normalized matching function among component name, GroupId, ArtifactId, package name, and CPE name. The output of this stage is not the final risk conclusion, but a candidate set for subsequent dynamic verification. Therefore, the method avoids directly equating “a component with a known CVE” with “an exploitable runtime risk”.

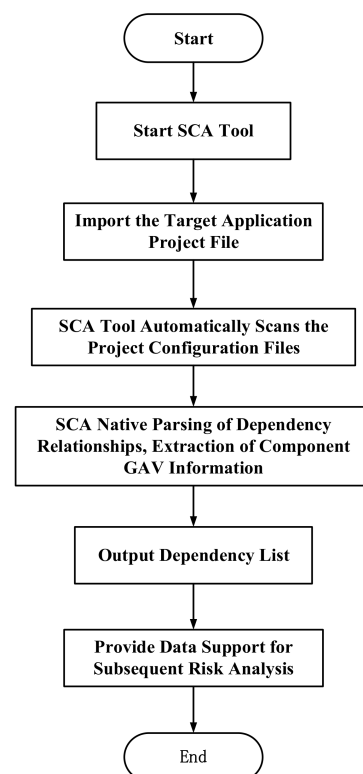


Figure 5. Dependency relationship extraction.

4.3. Static Vulnerability Identification

For each matched CVE record, SSCRPDMD parses its vulnerability description, CWE category, security advisory, patch information, and known exploitation chain to identify functions related to vulnerability triggering and propagation. This stage refines component-level CVE matching results into function-level risk objects for subsequent dynamic bytecode instrumentation [29].

Taking CVE-2021-44228 as an example, its description indicates that the affected product is the log4j-core component of Apache Log4j2. As shown in Figure 6, the original unstructured CVE description is parsed into a structured vulnerability tuple:

$$V_j = (ID_j, P_j, S_j, V_j^{affected}) \quad (12)$$

where ID_j denotes the CVE identifier, P_j denotes the affected product, S_j denotes the vulnerability description or semantic information, and $V_j^{affected}$ denotes the affected version range. The meaning of each field is shown in Table 2.

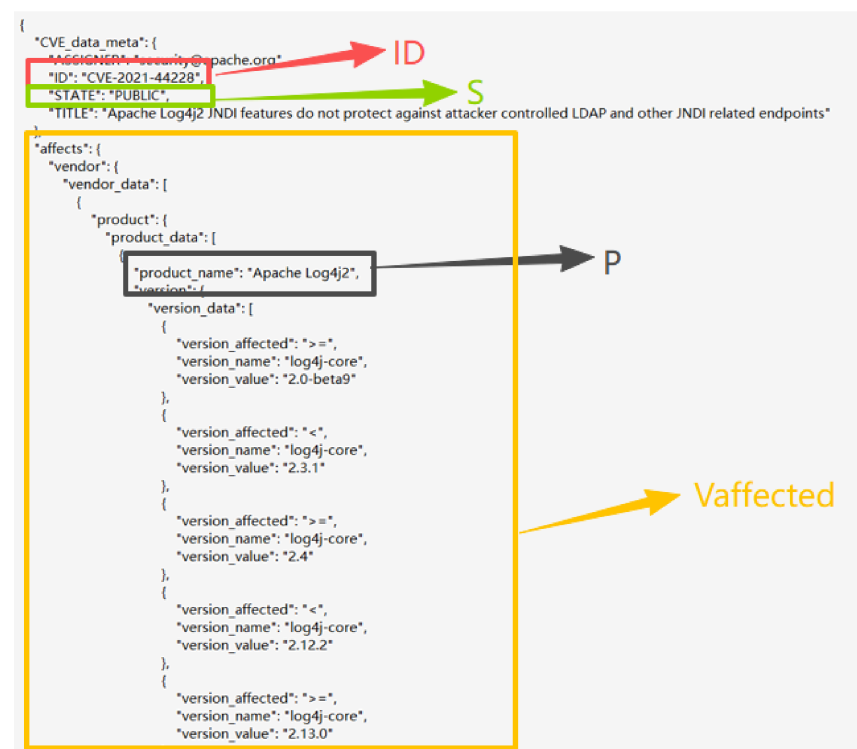


Figure 6. Description of CVE vulnerability.

Table 2. Explanation of Structure V.

Element	Meaning	Example
ID	Vulnerability identification	CVE-2021-44228
P	Target component	Apache Log4j2
S	Risk state	PUBLIC
Vaffected	Constrained version set	$V_{affected} = \{v \mid 2.0\text{-beta9} \leq v < 2.3.1\} \cup \{v \mid 2.4 \leq v < 2.12.2\} \cup \{v \mid 2.13.0 \leq v < 2.15.0\}$

To support function-level analysis, the structured CVE information is further transformed into a risk function representation:

$$\phi_j = (Entry_j, Sink_j, Cond_j, Type_j, Evi_j) \quad (13)$$

where $Entry_j$ denotes the possible trigger entry, $Sink_j$ denotes the risky function or dangerous operation, $Cond_j$ denotes the trigger condition, and $Type_j$ denotes the vulnerability type.

The extraction process integrates CVE semantic parsing, patch comparison, and bytecode confirmation to identify risk entries, sinks, trigger conditions, and sanitizer points. Only CVE records whose sink functions exist in the target component bytecode are retained for dynamic instrumentation.

The risk function set is defined as:

$$\Phi = \{\phi_j \mid (d_i, vul_j) \in R_0 \wedge Exists(Sink_j, d_i) = 1\} \quad (14)$$

where $Exists(Sink_j, d_i)$ indicates whether the corresponding sink function can be located in the bytecode of component d_i . Through this process, SSRPDM narrows the analysis scope from vulnerable components to concrete risk functions, providing a clear basis for minimal bytecode instrumentation.

Once a component containing a CVE is matched, it is not directly regarded as an exploitable risk. Instead, the vulnerability mechanism is further analyzed to identify the core risk functions that may trigger the vulnerability. In this paper, a risk function refers to a trigger entry, security-sensitive sink, or key method related to vulnerability exploitation, which is identified from CVE/NVD descriptions, public vulnerability reports, exploit mechanisms, official patches, and known call chains. Table 3 enumerates representative risk functions associated with common supply chain components [6,26].

Table 3. List of Risk Functions.

Risk Component	Triggering Entry	Core Risk Methods	Notes
Fastjson	JSON.parse() JSON.parseObject()	checkAutoType() deserialize()	leads to the invocation of Setter/Getter
XStream	XStream.fromXML()	TreeUnmarshaller.convert()	deserialization of XML to objects
Log4j-core	Logger.error() Logger.info()	javax.naming.Context.lookup() JndiManager.lookup()	whether the lookup parameter contains "ldap://"
Dom4j	SAXReader.read()	XMLReader.parse()	The underlying layer calls the parser of the JDK

Based on this definition, the extraction process first identifies the affected component and vulnerability type from CVE/NVD descriptions and public reports, then maps the vulnerability behavior to possible trigger entries and core risk methods, and finally verifies whether these methods exist in the bytecode of the matched component version. Only verified methods are retained as instrumentation targets. Therefore, the functions listed in Table 3 are not selected arbitrarily, but are determined according to vulnerability descriptions, exploit mechanisms, patch locations, and bytecode-level verification. These risk functions provide direct inputs for the dynamic instrumentation strategy in the next stage.

4.4. Runtime Reachability Analysis

After risk functions are extracted, SSRPDM uses a Java Agent to monitor the application during runtime [16]. During class loading, the Agent instruments only the methods related to the extracted risk functions, rather than the entire component or all methods.

As shown in Figure 7, when a risk-related interface is triggered, the system records its invocation context, method parameters, call stack, input source, sanitizer execution, and associated HTTP request entry. In the figure, the solid arrows indicate the results generated during the intermediate processing stages, whereas the dashed arrows indicate the final results. In this way, only components actually invoked during actual in-orbit business operations are regarded as runtime reachable [4] within the specific observation window. It should be noted that this empirical definition of reachability is premised on continuous monitoring within the actual production environment, rather than simulated testing environments.

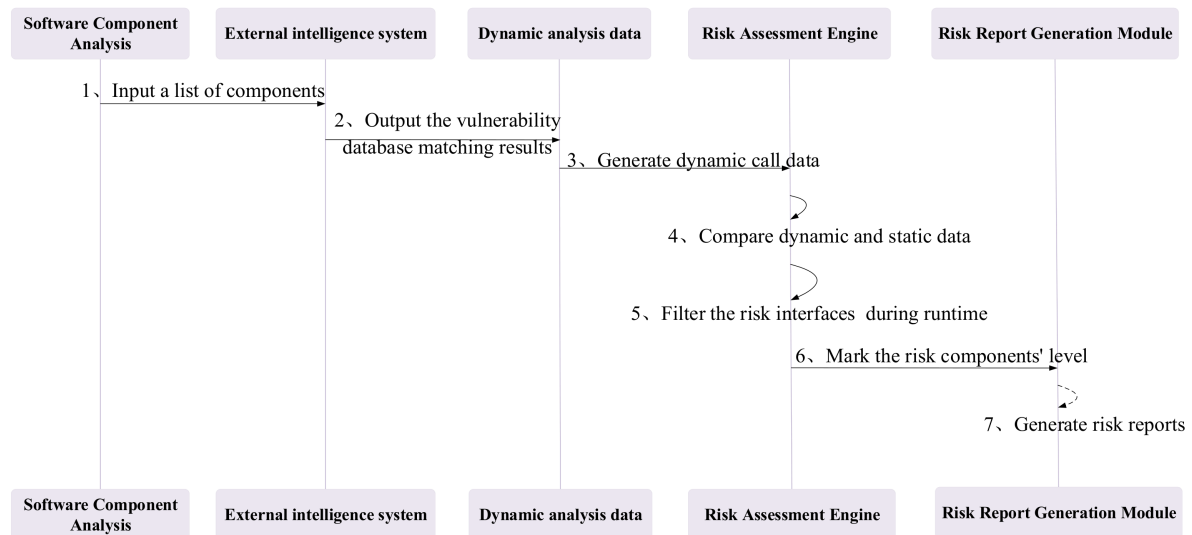


Figure 7. Risk interface analysis flowchart.

To make the instrumentation scope explicit, this paper defines a risk-function-guided minimal instrumentation strategy. For any method (m), the instrumentation decision is defined as:

$$I(m) = \begin{cases} 1, m \in Entry_j \cup Sink_j \cup Sanitizer_j \\ 1, m \in DangerousAPI \wedge Reachable(m, Sink_j) = 1 \\ 0, otherwise \end{cases} \quad (15)$$

where $Sanitizer_j$ denotes security filtering functions related to vulnerability mitigation, such as validation, escaping, filtering, protocol allowlist checking, and deserialization type restrictions. $DangerousAPI$ denotes system-level sensitive APIs.

Formula (15) restricts instrumentation to CVE-related entry, sink, sanitizer, and reachable dangerous API methods. Compared with full-component instrumentation, this policy reduces unnecessary bytecode enhancement and provides a formal basis for the lightweight runtime overhead evaluated in Section 5.6.

In implementation, the Java Agent instruments target methods through ClassFileTransformer and bytecode enhancement. Each probe generates a runtime record:

$$o_t = (rid, t, c, m, stack, param, src, taint, san, ret) \quad (16)$$

where rid denotes the request identifier, t denotes the timestamp, c denotes the component name, m denotes the invoked method, $stack$ denotes the call stack hash, $param$ denotes the desensitized parameter digest, src denotes the input source, $taint$ denotes whether external input propagates to the sink parameter, san denotes whether an effective sanitizer is executed before the sink, and ret denotes the return status or exception information.

Formula (16) defines a standardized runtime record, and the complete observation set is denoted by $\Omega = \{o_t\}$. This set serves as the unified data carrier connecting runtime instrumentation with subsequent reachability analysis, PoC verification, and LLM-based risk assessment.

Runtime reachability of vulnerability ρ_j is defined as:

$$\rho_j = \begin{cases} 1, \exists o_t \in \Omega, o_t.m \in Sink_j \wedge Path(Entry_j, Sink_j) = 1 \\ 0, otherwise \end{cases} \quad (17)$$

where $\rho_j = 1$ indicates that the corresponding risk function is triggered during runtime, while $\rho_j = 0$ indicates that the component exists in the dependency tree but the related risk function is not invoked within the observation period.

Formula (17) provides the vulnerability-level implementation of the reachability coefficient δ_{reach} introduced in Formula (1). When $\rho_j = 0$, the corresponding risk score RS_j is set to zero in Formula (21), thereby filtering out vulnerabilities whose risky execution paths are not observed at runtime.

The parameter controllability variable is further defined as:

$$\gamma_j = \begin{cases} 1, ExternalInput \rightarrow Param(Sink_j) \\ 0, otherwise \end{cases} \quad (18)$$

where $\gamma_j = 1$ means that external input can propagate to the parameter of the risk function. Only vulnerabilities satisfying $\rho_j = 1$ and $\gamma_j = 1$ are submitted to the subsequent PoC verification stage.

Formula (18) determines whether external input can influence the critical parameters of $Sink_j$. Since parameter controllability is a prerequisite for practical exploitation, γ_j is incorporated into the weighted risk score in Formula (21), with its contribution controlled by the weight λ_1 .

4.5. Exploitability Verification

In this stage, SSCRPDM verifies the exploitability of identified risk paths through controlled PoC replay [30]. The system replays the corresponding request entry in an isolated or authorized testing environment, injects CVE-specific payloads into externally controllable parameters, and observes whether the sink function is triggered and whether an observable exploitation effect is produced.

The PoC verification result is defined as:

$$\eta_j = \begin{cases} 1, Hit(Sink_j) = 1 \wedge Marker(Sink_j) = 1 \wedge Effect_j = 1 \wedge \sigma_j = 0 \\ 0.5, Hit(Sink_j) = 1 \wedge Marker(Sink_j) = 1 \wedge Effect_j = 0 \\ 0, otherwise \end{cases} \quad (19)$$

where $Hit(Sink_j)$ indicates that the sink function is triggered, $Marker(Sink_j)$ indicates that the PoC marker reaches the sink parameter, and $Effect_j$ denotes observable exploitation effects, such as callback behavior, command execution logs, exception stacks, file access, or database execution. σ_j denotes sanitizer effectiveness:

$$\sigma_j = \begin{cases} 1, Sanitizer_j \text{ is executed before } Sink_j \text{ and blocks malicious data} \\ 0, otherwise \end{cases} \quad (20)$$

The final risk score is calculated as:

$$RS_j = cvss_j \cdot \rho_j \cdot (\lambda_1 \gamma_j + \lambda_2 \eta_j + \lambda_3 (1 - \sigma_j)) \quad (21)$$

where ρ_j denotes runtime reachability, γ_j denotes parameter controllability, η_j denotes the PoC verification result, and σ_j denotes sanitizer effectiveness. λ_1 , λ_2 , and λ_3 are the corresponding weights and satisfy:

$$\lambda_1 + \lambda_2 + \lambda_3 = 1 \quad (22)$$

In this paper, $\lambda_1 = 0.3$, $\lambda_2 = 0.5$, and $\lambda_3 = 0.2$ are adopted. According to η_j and (RS_j) , risky components are classified into three levels: High Risk indicates that the PoC is successfully triggered and the payload reaches the sink without effective filtering; Medium Risk indicates that the path is reachable but the payload is intercepted, sanitized, or fails to produce an exploitable effect; Low Risk indicates that the path is unreachable or the sink cannot be triggered during verification.

4.6. LLM Analysis

Unlike existing LLM-based vulnerability assessment methods that directly input unstructured CVE descriptions or source code and rely on the model to independently infer risk, SSCRPDMM does not use the LLM as an autonomous vulnerability classifier. Instead, it introduces an evidence-constrained LLM analysis mechanism that organizes the outputs of static localization, runtime reachability analysis, parameter controllability analysis, sanitizer evaluation, and PoC verification into a unified structured input. The originality of this stage therefore lies not in the use of an LLM itself, but in the explicit coupling of deterministic security evidence with LLM-based semantic interpretation.

Given the complexity of call stacks and vulnerability data, this stage introduces a Large Language Model (LLM) as an auxiliary analysis engine [31]. Component versions, CVE descriptions, extracted risk functions, dynamic call stacks, PoC verification results, and partially desensitized business code snippets are constructed into a structured prompt context. In this paper, GPT-3.5 is used to analyze the vulnerability cause, invocation path, sanitizer status, and remediation suggestions [32].

For vulnerability vul_j , the input of the LLM analysis stage is defined as:

$$X_j = (vul_j, \phi_j, \Omega_j, \rho_j, \gamma_j, \eta_j, \sigma_j, RS_j) \quad (23)$$

where vul_j denotes the CVE record, ϕ_j denotes the extracted risk function, Ω_j denotes the runtime records related to vul_j , ρ_j denotes runtime reachability, γ_j denotes parameter controllability, η_j denotes the PoC verification result, σ_j denotes sanitizer effectiveness, and RS_j denotes the calculated risk score.

Equation (23) constitutes the first distinction from conventional LLM-based vulnerability analysis. Rather than allowing the LLM to infer exploitability directly from vulnerability descriptions, SSCRPDMM binds the analysis to function-level static evidence, runtime execution evidence, and PoC-based exploitability evidence. Thus, every LLM conclusion can be traced to a specific risk function, runtime path, controllability state, sanitizer state, and verification result.

The LLM input vector X_j encapsulates all core variables defined in Formulas (1)–(21), ensuring model analysis is fully bounded by deterministic runtime metrics rather than unconstrained text, solving the reproducibility defect of pure LLM vulnerability assessment.

To improve system performance during dynamic evaluation, a caching mechanism is introduced to avoid repeated analysis of identical call paths. The cache key is defined as:

$$K_j = \text{Hash}(id_j, Sink_j, stack_j, param_j, \rho_j, \gamma_j, \eta_j, \sigma_j) \quad (24)$$

where $stack_j$ denotes the call stack hash and $param_j$ denotes the desensitized parameter digest. If v already exists in the cache, the previous analysis result is reused.

The structured prompt is constructed as:

$$Prompt_j = \mathcal{P}(X_j, T) \quad (25)$$

where T denotes predefined analysis constraints, including reachable path, controllable parameter, effective sanitizer, successful PoC, and risk level rules. In particular, the prompt explicitly defines valid application-level sanitizers.

Equation (25) represents the second methodological distinction. The prompt is not generated solely from natural-language templates; instead, it is jointly determined by the evidence vector X_j and the formal constraint set T . Consequently, the LLM is required to explain the supplied evidence rather than generate an independent risk judgment without measurable support.

The LLM output is defined as:

$$Y_j = \mathcal{M}_\theta(Prompt_j, \tau) \quad (26)$$

where $\mathcal{M}_\theta(\cdot)$ denotes the LLM and τ denotes the temperature parameter. In this paper, $\tau = 0.2$ is adopted to reduce output randomness. The output is represented as:

$$Y_j = \left(Cause_j, Path_j, San_j, Fix_j, \tilde{L}_j \right) \quad (27)$$

where $Cause_j$ denotes the vulnerability root cause, $Path_j$ denotes the attack path analysis, San_j denotes the sanitizer analysis result, Fix_j denotes the remediation suggestion, and $\tilde{L}_j$ denotes the risk level suggested by the LLM.

The final risk level is determined by rule-based classification:

$$L_j = \begin{cases} High, \eta_j = 1 \wedge RS_j \geq \theta_H \\ Medium, \rho_j = 1 \wedge \gamma_j = 1 \wedge \eta_j < 1 \wedge RS_j \geq \theta_M \\ Low, \rho_j = 0 \vee \gamma_j = 0 \vee RS_j < \theta_M \end{cases} \quad (28)$$

where θ_H and θ_M are the thresholds for high-risk and medium-risk classification, respectively. The LLM-generated risk level L_j is used only as an auxiliary reference. If L_j conflicts with the rule-based result L_j , the rule-based result is retained. In this way, the LLM analysis stage converts complex call stacks, vulnerability descriptions, PoC results, and code contexts into readable reports, while keeping the final risk classification explicit and reproducible.

The detailed workflow of the LLM Semantic Evaluation is presented in Algorithm 1, which includes cache verification, semantic prompt construction, LLM inference, response parsing, and result caching. Based on this workflow, the innovation of this stage can be summarized as a three-part mechanism: (1) a cross-stage evidence vector that unifies function-level static information, runtime behavior, sanitizer status, and PoC results; (2) an evidence- and rule-constrained prompt that bounds LLM reasoning to verified security facts; and (3) a decision-decoupling strategy in which the LLM generates interpretable causes, paths, and remediation suggestions but cannot override the deterministic risk classifier. These mechanisms distinguish SSCRPDM from approaches that merely apply an LLM to vulnerability descriptions or source code and make the resulting analysis traceable, reproducible, and suitable for security-critical risk assessment.

Algorithm 1: LLM Semantic Evaluation**Input:** The Large Language Model interface, Callchain**Output:** RiskLevel

1. Cache Verification for Performance Optimization
2. CacheKey = GenerateHash(CallChain)
3. if Cache.contains(CacheKey) do
4. return Cache.get(CacheKey)
5. Semantic Prompt Construction
6. Prompt = "Analyze the following call chain for effective risk sanitization logic.\n"
7. Prompt = Prompt + "Valid Sanitizer Definition: Functions containing keywords such as 'sanitize', 'filter', 'clean', 'escape', or 'validate'.\n"
8. Prompt = Prompt + "Evaluation Criteria:\n"
9. Prompt = Prompt + "- HIGH Risk: Sink reached with no effective sanitizer.\n"
10. Prompt = Prompt + "- MEDIUM Risk: Sanitizer present but logic may be incomplete.\n"
11. Prompt = Prompt + "- LOW Risk: Valid and effective sanitizer executed before sink.\n"
12. Prompt = Prompt + "Call Chain Context:\n" + CallChain
13. LLM Inference
14. Temperature is set to 0.2 to ensure deterministic, analytical output 17: Response = LLM_Engine.generate(Prompt, temperature = 0.2)
15. Result Parsing and Caching
16. if Response **contains** "riskLevel: HIGH" **then**
17. RiskLevel = HIGH
18. **else if** Response **contains** "riskLevel: LOW" **then**
19. RiskLevel = LOW
20. **else**
21. RiskLevel = MEDIUM
22. **end**
23. Cache.put(CacheKey, RiskLevel)
24. **return** RiskLevel

5. Experiments and Analysis

In software-defined satellite systems and 6G non-terrestrial networks, JVM-based services can be deployed on satellite edge nodes, payload data processing gateways, and ground control systems to support functions such as telemetry processing, telecommand parsing, mission data forwarding, and secure logging. To evaluate this applicability, we built a space-oriented testbed using representative telemetry/telecommand workloads and intermittent link conditions, while constraining CPU and memory resources to simulate an edge deployment environment. Fastjson (v1.2.24) and Log4j (v2.11.0) were selected as reproducible third-party software supply chain vulnerability cases, rather than being claimed as satellite-specific vulnerabilities. In this context, SSCRPDm targets the same core software supply chain elements required by such systems: dependency resolution, runtime invocation, propagation of external inputs, and execution at security-sensitive sinks. Therefore, experimental results validate the application of SSCRPDm to JVM-based satellite edge software and ground control software under space mission operational constraints.

5.1. Dataset Construction

The experiment used the Trickest CVE repository [33] as the reference CVE and proof-of-concept (PoC) dataset. The repository collects CVE information, organizes the records by year, and identifies publicly available PoCs through CVE reference links and GitHub repositories. The local snapshot used in this study contained more than 300,000 CVE records covering the period from 1999 to 2025. Its website and access date are provided as follows: Available online: <https://github.com/trickest/cve>.

The Goblin Maven Central dependency graph dataset (version 2024-08-30) [34], provided as the `goblin_maven_30_08_24.dump` file, was utilized to query component dependency relationships within Maven projects. Basic nodes and relationships—such as Artifact, Release, and dependency—were validated. Furthermore, Cypher queries were employed to confirm the dependency relationships of specific components, such as `fastjson` and `log4j-core`. The dataset construction proceeded through the following steps:

Step 1: Search for the target component ID.

Step 2: Query dependencies based on the target component ID information.

Step 3: Consolidate the queried dependency chains to form the dataset.

Figure 8 illustrates the dependency relationships of the target component “`org.apache.logging.log4j:log4j-core`,” including metadata such as component ID, version, and timestamp. The `[:dependency]` edges indicate the component’s dependencies on downstream elements, with annotated dependency scopes. For the specific version `log4j-core:2.11.0`, the system not only identifies the component itself but also fully maps its dependencies on downstream artifacts such as `commons-io` and `jmdns`, along with their respective scopes, via the defined dependency relationships.

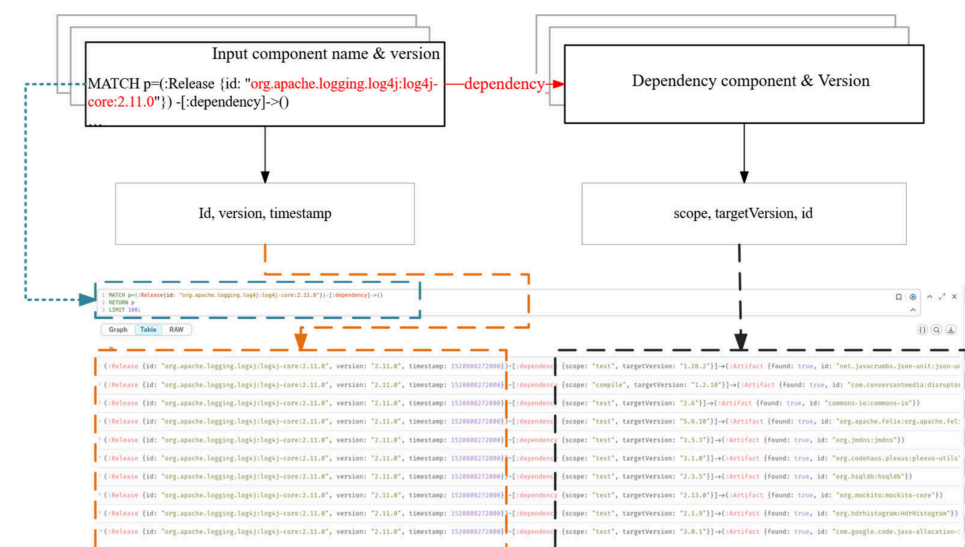


Figure 8. Component dependency query results.

5.2. Experimental Workflow

- SCA Scanning

In Maven projects, dependency version information is explicitly declared in the `pom.xml` file, and the complete dependency tree can be retrieved by executing the `mvn dependency:tree` command. Consequently, the SCA tool is employed to scan the project’s `pom.xml` file or the generated dependency tree to identify components containing CVEs; these are subsequently defined as risky components. Upon scanning the application containing the `Fastjson` component, the results are illustrated in Figure 9, which revealed a total of 50 detected CVEs. The star symbol is defined by the Dependency-Check tool and indicates

that the corresponding result may be a potential false positive. Therefore, vulnerabilities marked with a star cannot be directly regarded as confirmed vulnerabilities and must be further verified according to the actual runtime invocation and execution conditions.

Summary

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
fastjson-war: commons-io-2.3.jar	cpe:2.3:a:apache:commons-io:2.3:*:*:*	pkg:maven/commons-io/commons-io@2.3	MEDIUM	2	Highest	108
fastjson-war: fastjson-1.2.24.jar	cpe:2.3:a:alibaba:fastjson:1.2.24:*:*:*	pkg:maven/com.alibaba.fastjson/fastjson@1.2.24	CRITICAL	2	Highest	37
fastjson-war: servlet-api-6.0.32.jar	cpe:2.3:a:apache:tomcat:6.0.32:*:*:*	pkg:maven/org.apache.tomcat/servlet-api@6.0.32	CRITICAL*	46	Highest	20

Figure 9. SCA scanning results.

- **Reachability Analysis**

Based on the identified risky components, a bytecode instrumentation Agent is deployed to monitor the key risk methods of the target components. Distinct risk methods are associated with specific risky components.

- **PoC Verification**

For triggered risk paths, specific payloads are constructed to execute replay attacks. If the instrumentation logs capture the malicious payload without evidence of sanitization, the risk is adjudicated as a confirmed threat. The overall process is illustrated in Figure 10.

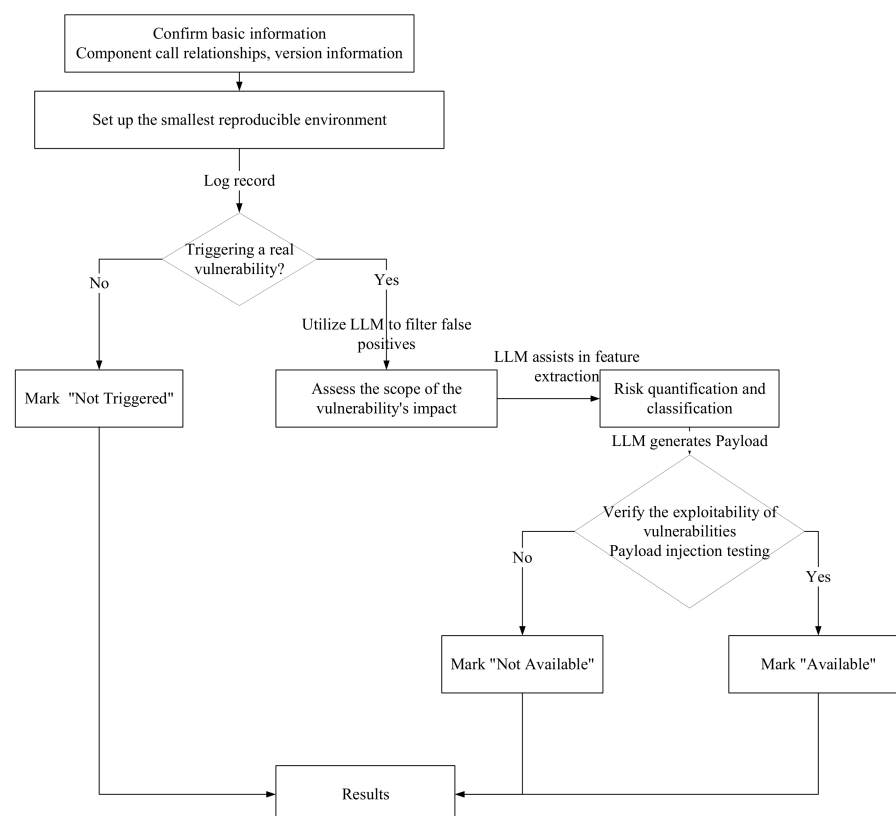


Figure 10. Overall flowchart of the experiment.

5.3. Process Implementation

In the target application, the fastjson component was detected, and SCA scanning revealed a deserialization risk associated with it. Bytecode enhancement was subsequently applied to the component's JSON.parse() method. Figure 11. displays the runtime logs, which indicate that the probe successfully identified the target server environment, detected

server astomcat/9.0.100.0 during the class loading phase, and performed hooking operations on critical classes using the Java Instrumentation API. The log entry 'insert before method java.lang.UNIXProcess' demonstrates that the system successfully implanted monitoring logic at the entry point of the underlying process creation method, a prerequisite for capturing Remote Code Execution (RCE) vulnerabilities. Additionally, the logs record the instrumentation of java.io.FileInputStream and com.mysql.cj.jdbc.StatementImpl, verifying that the system possesses comprehensive runtime awareness regarding file operations and SQL execution.



Figure 11. Fastison alarm log details.

Upon confirming the readiness of the monitoring environment, we proceeded to the PoC verification phase. In the experiment, a malicious JSON payload targeting the Fastjson TemplatesImpl gadget chain was sent to the target range to trigger remote command execution.

The logs indicate that the request was received by `person.FastJsonServlet.doGet` and processed via the business logic `person.Poc.testAutoTypeDeny`. Subsequently, the data flowed into `JSON.parseObject`, the core parsing method of the Fastjson component. This confirms that the component was indeed loaded and invoked within the business logic, verifying the reachability of the risk. Simultaneously, the exploitability of the vulnerability was verified. During the deserialization process, Fastjson triggered the `com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl` gadget chain. Specifically, the `getOutputProperties` method invoked `newTransformer`, which subsequently instantiated the malicious class `person.Test` constructed by the attacker via the reflection mechanism `Constructor.newInstance`. At the top of the call stack, `person.Test.<init>` was executed, attempting to invoke `java.lang.ProcessBuilder.start` and `java.lang.UNIXProcess.<init>`. Consequently, the execution path of the risky component is inferred as shown in Figure 12.

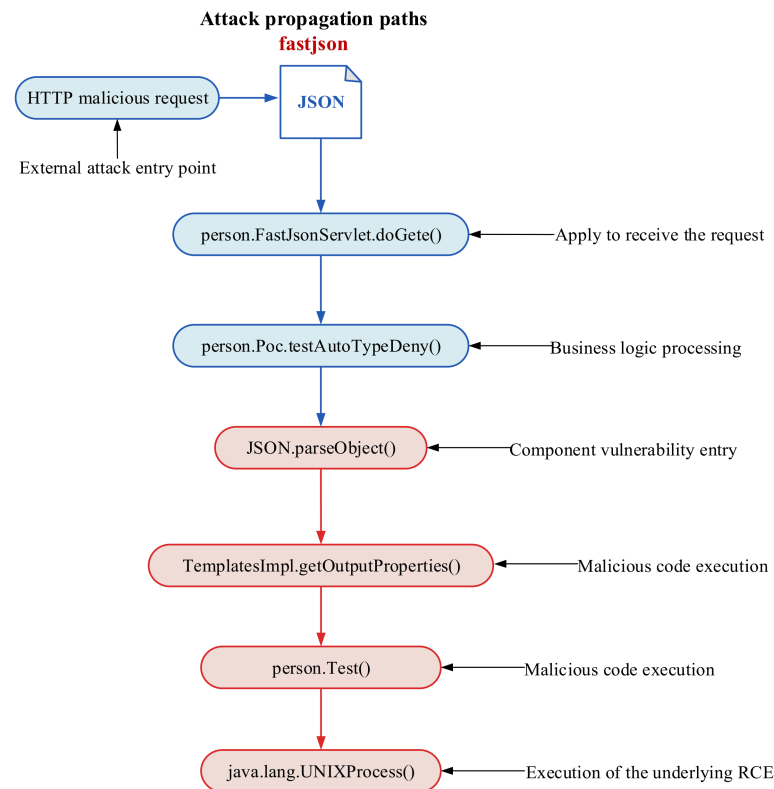


Figure 12. Fastjson invocation path.

5.4. Result Analysis

In the experimental analysis, Various indicators such as Precision is used to reflect the effect of SSCRPDM from different perspectives. Precision measures the overall degree of correct prediction by the model [35]. Precision is calculated using Equation (29), and is mainly related to the number of correct detections by the method.

$$Precision = \frac{TP}{TP + FP} \quad (29)$$

TP: True Positive, the component does indeed have vulnerabilities and they have been successfully exploited, and the system correctly raised an alarm;

FP: False Positive, components are introduced but not invoked, or invoked but filtered by security mechanisms (with no actual harm), yet the system wrongly raises an alarm.

Upon completion of the experimental testing and the analysis of invocation chain logs, the execution results were summarized and analyzed. Using the JNDI injection vulnerability as a case study, Table 4 presents the number of components detected at each stage by the proposed method.

An analysis of the table reveals a dramatic funnel effect in filtering false positives, highlighted by a sharp increase in detection precision. Figure 13 shows, while traditional SCA reported 50 risky components—yielding a precision rate of only 6%—dynamic reachability analysis successfully excluded 76% of these initial alerts that were not utilized at runtime, raising the precision to 25%. Ultimately, PoC verification confirmed that only three components possessed genuine exploitability within the current business context, achieving a flawless 100% precision. It can be concluded that these components are not only present in the dependency list but are also reachable during actual business operations and lack filtration for malicious inputs. Consequently, their risk level is definitively escalated to ‘Critical’.

Table 5 presents the corresponding EPSS for the CVE vulnerabilities. To further analyze the relationship among CVSS, RS, and EPSS, the Pearson correlation coefficient was introduced to quantify the correlation among the three.

Table 5. The corresponding EPSS for the CVE vulnerabilities.

CVE ID	CVE-2024-47554	CVE-2021-29425	CVE-2017-18349	CVE-2022-25845	CVE-2016-8735	CVE-2016-5018	CVE-2016-0714	CVE-2016-5388	CVE-2014-0230	CVE-2020-8022
EPSS(%)	0.17	0.62	88.9	88.57	93.67	0.93	6.01	36.76	4.9	0.19
CVE ID	CVE-2011-3190	CVE-2013-2185	CVE-2016-6796	CVE-2016-6797	CVE-2017-5647	CVE-2016-6816	CVE-2013-2067	CVE-2013-4444	CVE-2014-0227	CVE-2016-0762
EPSS(%)	0.87	5.29	0.82	0.56	4.15	2.78	10.45	20.29	79.83	0.72
CVE ID	CVE-2013-4286	CVE-2015-5345	CVE-2016-6794	CVE-2011-1184	CVE-2011-2729	CVE-2011-3375	CVE-2011-4858	CVE-2011-5062	CVE-2012-0022	CVE-2012-2733
EPSS(%)	27.07	14.82	0.36	2.24	8.78	2.02	76.60	5.32	23.42	20.28
CVE ID	CVE-2012-3544	CVE-2012-5885	CVE-2012-5886	CVE-2012-5887	CVE-2014-0075	CVE-2014-7810	CVE-2011-2526	CVE-2011-5063	CVE-2011-5064	CVE-2012-3546
EPSS(%)	44.77	2.27	0.57	0.90	52.26	9.49	0.13	1.96	5.32	2.23
CVE ID	CVE-2012-4431	CVE-2013-4322	CVE-2013-4590	CVE-2014-0096	CVE-2014-0099	CVE-2014-0119	CVE-2015-5174	CVE-2016-0706	CVE-2012-4534	CVE-2011-2204
EPSS(%)	9.82	48.49	0.92	5.79	39.08	4.35	3.69	1.54	22.77	0.10

The Pearson correlation coefficient [36] between two variables is defined as the ratio of the covariance between the two variables to the standard deviation:

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \quad (30)$$

This formula defines the overall correlation coefficient, and commonly uses Greek lowercase letters as the representative symbols. By estimating the covariance and standard deviation of the sample, the Pearson correlation coefficient can be obtained, which is represented by the common English lowercase letter:

$$r = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \bar{Y})^2}} \quad (31)$$

A three-dimensional correlation analysis was conducted on the CVSS, EPSS and RS scores of 50 CVE samples. Figure 15 presents the two-dimensional correlation between CVSS and RS. Each dot represents one CVE sample, with its horizontal and vertical coordinates corresponding to the CVSS and RS scores, respectively. The dashed line is the linear trendline automatically to calculate the coefficient of determination. The two-dimensional scatter plot analysis revealed that Figure 15 shows a Pearson correlation coefficient $r = 0.481556$ between CVSS and RS, indicating a significant moderate positive correlation, suggesting a positive association between the inherent hazard of the vulnerability and the real-time risk.

However, there are many discrete samples, indicating that RS introduces an additional risk dimension. Figure 16 shows the correlation between CVSS and EPSS, with $r = 0.325269$, indicating a significant weak positive correlation, reflecting the difference in assessment perspectives between the static technical hazard and the dynamic utilization probability. Figure 17 shows the correlation between RS and EPSS, with $r = 0.9232$, being the strongest among the three groups of correlations, indicating that RS highly integrates the vulnerability exploitability represented by EPSS and the real-time threat intelligence, and is more in line with the actual combat scenario.

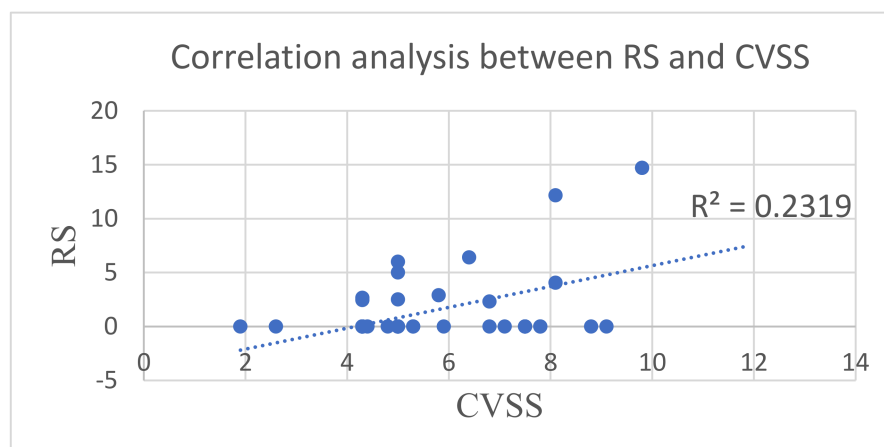


Figure 15. Correlation analysis between RS and CVSS.

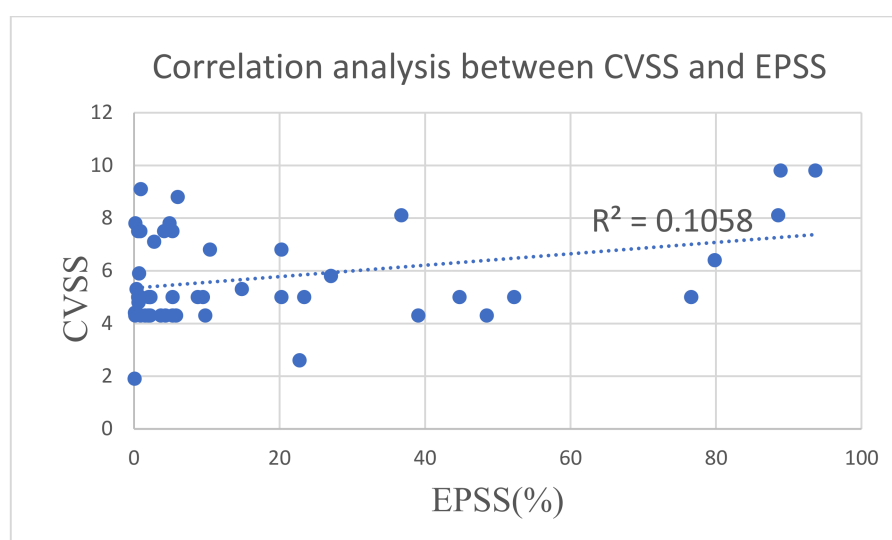


Figure 16. Correlation analysis between CVSS and EPSS.

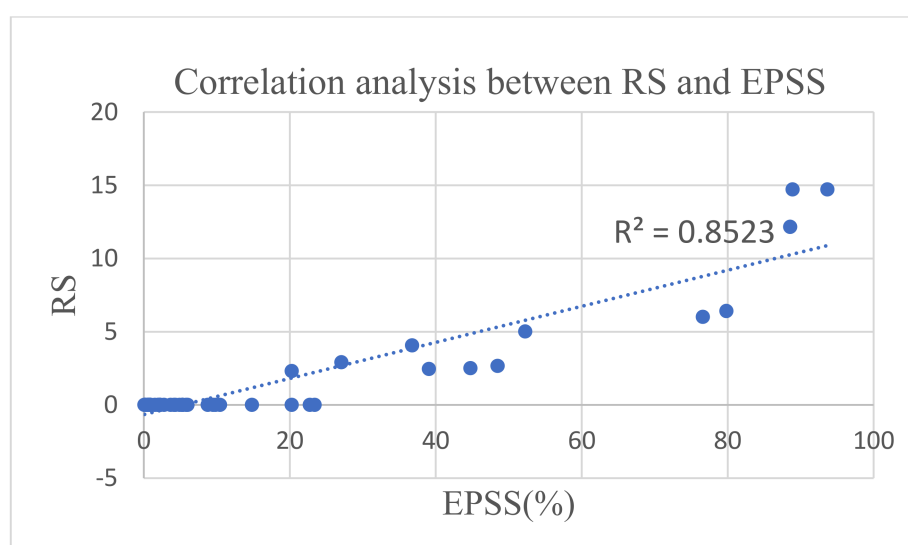


Figure 17. Correlation analysis between RS and EPSS.

In conclusion, CVSS, EPSS and RS have significant complementarity: CVSS establishes the baseline of inherent hazards, EPSS quantifies the utilization probability, RS integrates

multiple dimensions to form a comprehensive and real-time risk assessment, and combined use can achieve precise vulnerability priority ranking, providing a scientific basis for security resource allocation.

The experiment validates the correctness of the closed-loop detection logic. The call stacks within the alert logs demonstrate that relying solely on static version matching is insufficient to characterize risk; only by integrating runtime invocation chains (Call Stacks) can the substantive hazard of a vulnerability be confirmed. Unlike traditional scanning tools that merely report the presence of a Fastjson vulnerability, SSCRPDm provides precise ‘line-level’ risk localization by capturing runtime execution and specific system commands. Furthermore, this study incorporates a Large Language Model (GPT-3.5) to perform automated assessment and analysis of experimental data, thereby enhancing the intelligence and precision of the risk assessment.

5.5. Runtime Validation on the Yamcs System

To validate the applicability of SSCRPDm in a realistic space mission context, we constructed a simulation environment using Yamcs, a prominent open-source Mission Control System implemented in Java. Yamcs is widely utilized for payload testing, satellite telemetry processing, and telecommand execution. We deployed Yamcs as a simulated ground control gateway. The SSCRPDm Java Agent was seamlessly attached to the Yamcs JVM instance. To simulate dynamic orbital operations, a continuous workload of standard space communication protocols was generated, feeding telemetry (TM) streams at 100 Hz and sporadically issuing telecommand (TC) requests. We selected typical vulnerable dependencies within the Yamcs ecosystem (e.g., specific versions of Netty for network I/O or SnakeYAML/Jackson for configuration parsing) as our targets to simulate a supply chain attack via maliciously crafted TC payloads.

As shown in Figure 18, during the benchmark testing in the Yamcs aerospace mission control environment, traditional SCA exposed 142 potential vulnerabilities, leading to severe alert fatigue. By introducing the SSCRPDm framework, the dynamic reachability analysis precisely isolated and stripped away zombie components that were not invoked by the core TM/TC business execution chains, drastically reducing the risk baseline to 18 items, which are the component set where $\rho_j = 1$. The final PoC verification stage further confirmed that only 2 vulnerabilities were not blocked by effective sanitizers ($\eta_j = 1$), thereby achieving a precise convergence from massive static noise to a very few critical threats.

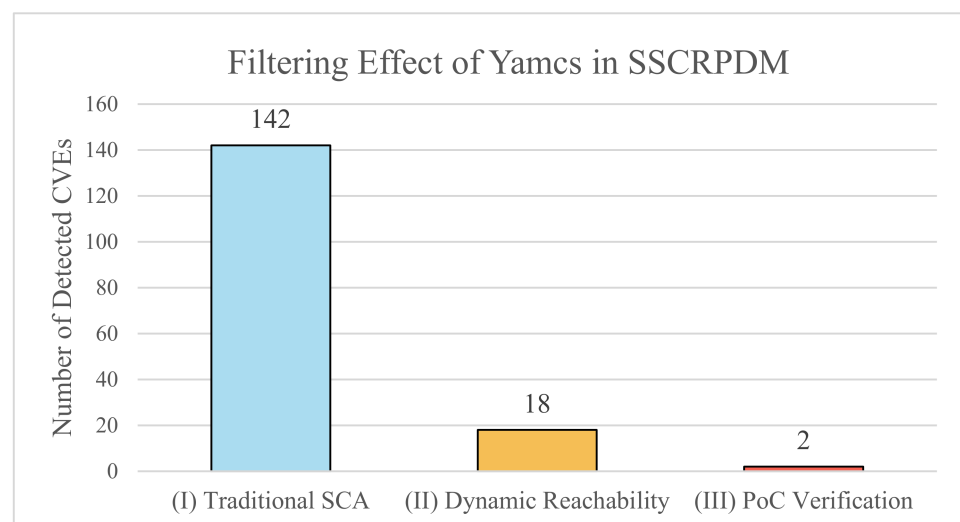


Figure 18. Yamcs verification results.

Yamcs incorporates a massive dependency graph to support various satellite plugins and database backends. Traditional SCA reported numerous candidate CVEs across the entire platform. However, under the specific mission profile, SSCRPDMD's dynamic reachability analysis confirmed that over 85% of these vulnerable components were zombies never invoked by the core telemetry processing threads, drastically reducing alert fatigue for ground operators.

5.6. Resource Expenditure

The payload of space communication satellites is subject to strict SWaP (size, weight and power) limitations, and its computing and storage resources are extremely scarce. To assess the impact of SSCRPDMD on the in-orbit standard service performance, this section simulates an in-orbit resource-constrained environment and compares the performance of systems with and without enabling SSCRPDMD when processing standard space communication protocol requests. Table 6 details the resource consumption of the system during the continuous 60 s processing of standard space communication service requests.

Table 6. Resource consumption comparison with and without SSCRPDMD deployment.

Time (s)	CPU Utilization Rate		Memory Occupancy (MB)	
	No SSCRPDMD	Open SSCRPDMD	No SSCRPDMD	Open SSCRPDMD
0	25.5	28.56	3.06	119.45
6	31.18	33.93	2.75	121.28
12	29.96	34.87	4.91	121.35
18	23.33	35.05	11.72	123.57
24	21.99	31.78	9.79	123.84
30	22.01	32.55	10.54	124.32
36	20.86	29.87	9.01	125.51
42	20.13	29.96	9.83	125.96
48	19.54	26.87	7.33	126.52
54	20.37	24.83	4.46	126.79
60	19.14	22.75	3.61	127.49

Figures 19 and 20 present the change curves more intuitively. In the baseline state without the instrumentation mechanism, due to the tidal changes of normal business loads, the CPU utilization of the system fluctuates dynamically between 19% and 31%, while the memory usage remains stable at approximately 119 MB to 128 MB. After deploying the proposed SSCRPDMD dynamic instrumentation mechanism, the system still maintains a highly stable operating state. The analysis of key performance indicators indicates that the additional CPU overhead introduced by SSCRPDMD is on average 7.01%, and the absolute increment in memory consumption is strictly controlled within approximately 9.2 MB. This continuous slight performance loss fully proves that this mechanism adopts an efficient on-demand detection strategy, not only does it not cause substantial interference to the execution efficiency of the main business, but also demonstrates its outstanding lightweight advantages and engineering deployment feasibility in the highly scarce computing and storage resources of the vehicle-mounted environment.

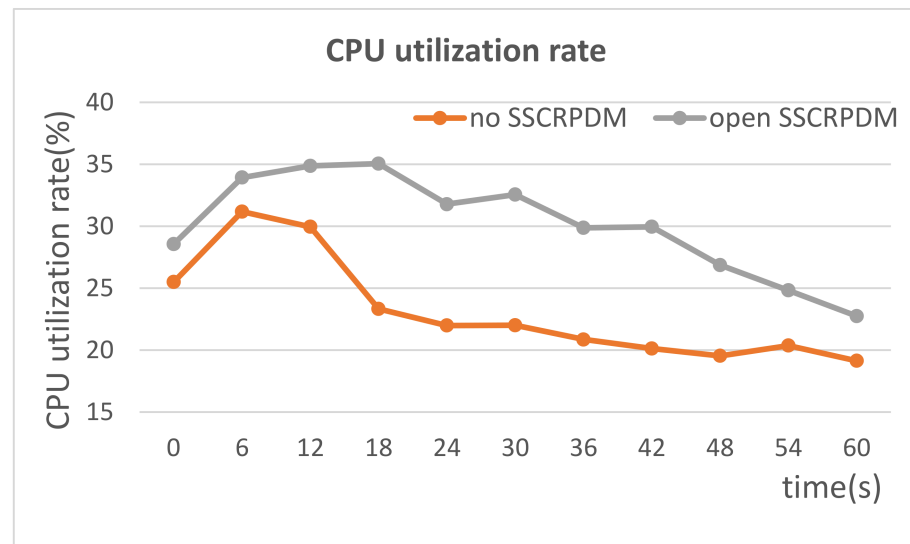


Figure 19. Comparison of CPU utilization rate.

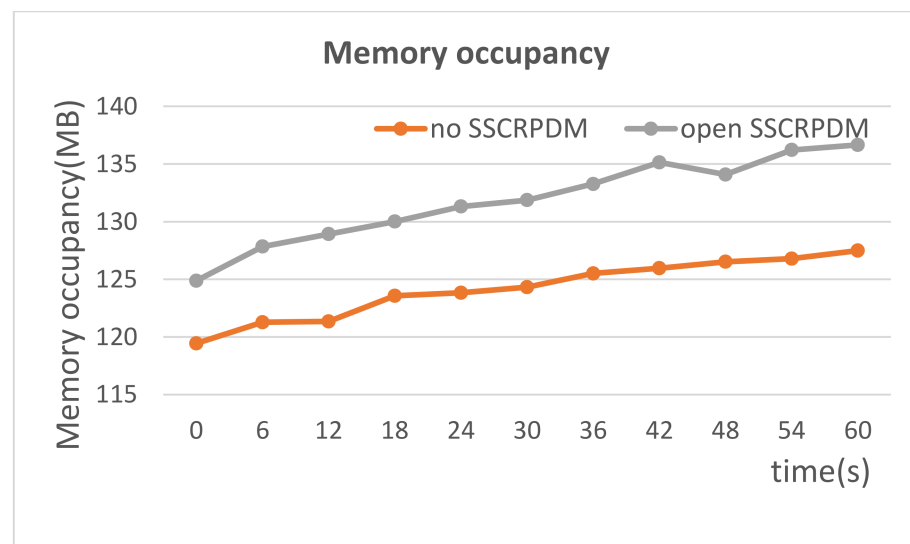


Figure 20. Comparison of memory occupancy.

5.7. Accuracy of LLM Engine

In this experiment, we used LLM to automatically parse the captured system operation log “alarm.log” and the CVE List related to the project. The log recorded a typical injection attack targeting the downstream components of the software supply chain.

Triggering source: The attacker initiates a malicious HTTP GET request through person.FastJsonServlet.doGet.

Vulnerability component chain: The request proceeds to the com.alibaba.fastjson library for deserialization processing. LLM identified that the call to com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl in the stack is the core exploitation point of this vulnerability, and it triggered illegal class instantiation through the getOutputProperties method.

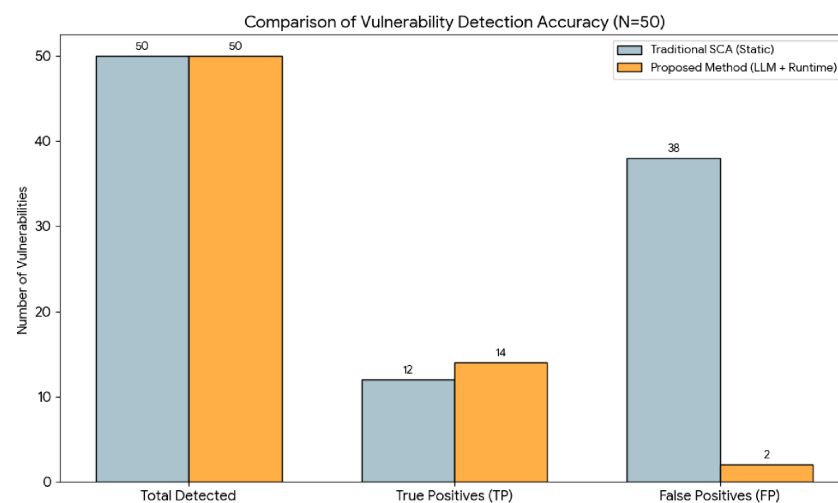
Execution outcome: The attacker successfully exploited the reflection mechanism to bypass security restrictions and ultimately invoked java.lang.ProcessBuilder.start to execute the system command ps-ef.

LLM, based on the reachability of the runtime call stack, has conducted a refined classification of the involved CVE vulnerabilities. The results are presented in the following Table 7.

Table 7. Vulnerability Risk Classification Table Based on Runtime Analysis.

CVE Number	Running Status	Risk Level	LLM Assessment Basis
CVE-2017-18349	Active	Critical	The call stack clearly shows that the parseObject triggered the illegal instantiation of a class and executed a system command.
CVE-2022-25845	Potential	Medium	Although the version number is shown as 1.2.x compatible chain, its autoType bypass logic is consistent with the current calling path.
else	Inactive	Low	Some vulnerabilities that exist in the dependency tree did not appear in this execution call chain.

To quantitatively evaluate the effectiveness of the large model-driven dynamic analysis method, the experiment selected 50 known CVE vulnerabilities from the project as the sample set. The experimental results are shown in Figure 21.

**Figure 21.** Comparison of the accuracy of vulnerability identification between traditional SCA and LLM methods.

When dealing with the same 50 potential risk points, the large model demonstrated extremely high discrimination ability. Traditional SCA tools, due to their inability to perceive the reachability of the code, generated as many as 38 false alarms in this experiment. However, the research method in this study, by understanding the call chain semantics through the large model, marked components with known vulnerabilities as low risk if they were not present in the stack, thereby reducing 94.7% of invalid alerts. At the same time, LLM can identify cross-component exploitation chains, and this deep semantic correlation is not possessed by traditional version number matching tools.

To evaluate the LLM stage quantitatively rather than anecdotally, we treated the PoC-verified risk level of each of the 50 CVE samples as ground truth and compared it against the LLM-assigned level. Using the three-level labels (High/Medium/Low) collapsed to a binary exploitable/non-exploitable decision, the LLM classifier achieved a precision of 0.95, a recall of 0.92, and an F1-score of 0.93, with a single false negative (a medium-risk sanitized path that the model over-trusted) and one false positive among the 50 samples; the macro-averaged accuracy over the three-level task was 94%. Table 8 summarizes the confusion between the LLM verdict and the PoC ground truth. We also assessed the 12 remediation snippets that the model generated for the high-risk findings: 10 of 12 (83%) compiled and applied without modification and correctly closed the verified data flow (e.g., enabling Fastjson SafeMode, removing the unused xalan transformer), while 2 required minor manual adjustment of version constraints. These figures quantify both the benefit

and the residual error rate of the LLM stage and motivate retaining the deterministic PoC verdict as the authoritative signal.

Table 8. Confusion Matrix of LLM Vulnerability Risk Classification vs. PoC Ground Truth ($N = 50$).

LLM Prediction	High Risk (Predicted)	Medium Risk (Predicted)	Low Risk (Predicted)
High Risk (Predicted)	2	1	0
Medium Risk (Predicted)	0	8	1
Low Risk (Predicted)	0	0	38

LLM automatically generated protection suggestions for the identified high-risk vulnerabilities:

- In response to the `command_reflect` attack behavior identified in the logs, it is recommended to implement interception policies in the RASP (Runtime Application Self-Protection) module, and prohibit `fastjson` from triggering `ProcessBuilder`.
- It is recommended to urgently upgrade `FastJson` to version 1.2.83 or above, and globally enable `SafeMode` to completely disable the `AutoType` function.
- During the build phase (CI/CD), remove the no-longer-used `xalan.jar` and related vulnerable Transformer auxiliary classes to reduce the attack surface.

6. Discussion

6.1. Applicability and Runtime Coverage

The proposed SSCRPDMD is mainly applicable to software systems whose runtime behavior can be captured through dynamic instrumentation. In this paper, the method is implemented for Java-based applications using the Java Instrumentation API, making it suitable for JVM-based satellite payload software, ground-control services, and Web applications with complex third-party dependencies. Its key advantage lies in converting static dependency risks into function-level runtime evidence without modifying the original business logic. For native binaries or statically compiled programs, future work may extend SSCRPDMD with binary rewriting or dynamic tracing mechanisms. Moreover, the core prerequisite for our reachability determination is that the system operates under a mission-representative workload over a sufficient observation period in the real-world environment. We acknowledge the inherent limitation of dynamic instrumentation regarding path coverage. If a vulnerable function resides in a rarely executed branch (e.g., edge-case exception handling or specific periodic maintenance tasks) that is not triggered during the continuous observation window, it will be classified as unreachable ($\rho_j = 0$). This could potentially lead to false negatives, where vulnerable code is misjudged as safe simply because its execution conditions were not met during the monitoring phase. To mitigate this, SSCRPDMD is designed for continuous, long-term on-orbit monitoring to progressively accumulate execution coverage of actual business logic, rather than relying on the limited coverage of pre-deployment testing. Since reachability depends on the workload executed within the observation window, constructing mission-representative workloads for rare satellite operation modes will be an important direction for further improving the method.

6.2. Comparison with Existing Detection Methods

Compared with traditional SCA, SSCRPDMD goes beyond component-version matching and further verifies whether the risky function is actually invoked at runtime. Compared with SAST, it avoids relying only on static code patterns and reduces false positives caused

by infeasible paths. Compared with DAST, it provides more precise localization of vulnerable components, risk functions, and invocation chains, rather than only reporting black-box attack symptoms. Compared with IAST, SSCRPDm further introduces CVE-driven risk-function extraction and PoC-based exploitability verification, producing a graded risk result based on reachability, parameter controllability, sanitizer effectiveness, and PoC results. Therefore, its innovation lies not in simply combining existing tools, but in linking static vulnerability discovery with runtime exploitability confirmation at the function level.

6.3. Reproducibility of LLM-Assisted PoC Verification

The LLM module improves the analysis capability of SSCRPDm, especially in understanding complex call stacks, identifying possible sanitizer logic, and assisting payload construction. However, LLM-based generation may vary with model versions and inference settings, making strict reproducibility difficult to fully guarantee at the current stage. To reduce such variation, SSCRPDm adopts fixed prompt templates, predefined output formats, unified classification criteria, and low-temperature inference. More importantly, the final risk score is not determined by the LLM output alone. Any generated payload must be checked through runtime instrumentation. If the payload cannot reach the risk sink or produce observable exploitation effects, the PoC is treated as failed and cannot increase the risk level. Therefore, SSCRPDm uses runtime verification to improve the reliability of LLM-assisted analysis. How to further improve the stability and repeatability of LLM-assisted PoC verification remains an important direction for future research.

7. Conclusions

The study successfully achieves the precise detection and impact assessment of risky components within the software supply chain. We proposed and implemented an innovative method named SSCRPDm, a closed-loop risk assessment framework that integrates SCA version detection, dynamic reachability analysis, and automated PoC verification. This method effectively identifies high-risk components that are actually invoked and maps their vulnerability trigger paths.

Compared to traditional tools that rely solely on static or dynamic analysis, SSCRPDm leverages bytecode instrumentation to filter out ‘zombie components’—those introduced but not loaded. Furthermore, it employs PoC verification to eliminate invalid risks that are intercepted or filtered by business logic, achieving a precise convergence from ‘theoretical risk’ to ‘confirmed threat.’ Experimental results demonstrate that code reachability is not equivalent to exploitability; a vulnerability can only be confirmed as critical if PoC verification proves that the data flow remains unsanitized at the point of risk. Through simulation experiments in the Yams space mission control system, it was further demonstrated that this method is feasible in real space communication scenarios and has extremely low performance overhead.

Additionally, this paper utilizes LLMs to automatically generate version-specific verification payloads by combining CVE descriptions with dynamic call chains. This enhances the precision of PoC verification for specific components while reducing manual labor costs. Future work will aim to overcome the limitations of CVE databases by focusing on the discovery of unknown (zero-day) vulnerabilities. By monitoring the anomalous triggering behaviors of underlying sensitive methods, we aim to achieve the automated mining of potential unknown vulnerabilities in supply chain components, thereby further enhancing the system’s active defense efficacy.

Author Contributions: Conceptualization, R.G., N.H. and G.L.; methodology, R.G. and N.H.; software, R.G., Z.H. and F.P.; validation, R.G., N.H., Y.M. and Z.H.; formal analysis, R.G. and Y.M.; investigation, R.G., N.H. and F.P.; resources, G.L. and G.W.; data curation, R.G. and Z.H.; writing—

original draft preparation, R.G.; writing—review and editing, N.H., G.L. and G.W.; visualization, R.G. and F.P.; supervision, G.L. and G.W.; project administration, G.L. and G.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by NSFC under Grant U23B2021.

Data Availability Statement: The CVE data analyzed in this study are publicly available in the Trickest CVE repository [33]. The Maven component dependency data were obtained from the Goblin Maven dependency graph dataset [34]. The processed data and experimental results generated during this study are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zimmermann, M.; Staicu, C.-A.; Tenny, C.; Pradel, M. Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*; USENIX Association: San Francisco, CA, USA, 2019; pp. 995–1010.
2. Ohm, M.; Plate, H.; Sykosch, A.; Meier, M. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Lisbon, Portugal, 24–26 June 2020; pp. 23–43.
3. Banks, R.; Jones, J.; Hazzazi, N.; Garcia, P.; Zimmermann, R. Hybrid Security Risk Assessment Model. In *Advances in Intelligent Systems and Computing*; Springer International Publishing: Cham, Switzerland, 2021; pp. 77–83.
4. Pashchenko, I.; Plate, H.; Ponta, S.E.; Sabetta, A.; Massacci, F. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'18)*, Oulu, Finland, 11–12 October 2018; pp. 1–10.
5. Soto-Valero, C.; Durieux, T.; Harrand, N.; Baudry, B. Coverage-Based Debloating for Java Bytecode. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–34. [\[CrossRef\]](#)
6. Zapata, G.A.; Katerina, G.; Sewe, A. Towards automated library vulnerability assessment with static analysis. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, Italy, 20–23 March 2018; pp. 562–566.
7. Kula, R.G.; German, D.M.; Oentaryo, R.J.; Lo, D. Do developers update their library dependencies? *Empir. Softw. Eng.* **2018**, *23*, 384–417.
8. Qi, A.X. 2025 China Software Supply Chain Security Analysis Report. Available online: https://www.qianxin.com/news/detail?news_id=13666 (accessed on 7 July 2025).
9. Ponta, S.E.; Plate, H.; Sabetta, A.; Bezzi, M. Detection of known vulnerabilities in software dependencies: An industrial case study. In *Proceedings of the 42nd International Conference on Software Engineering: Software Engineering in Practice*, Seoul, Republic of Korea, 19–27 June 2020; pp. 201–210.
10. Xu, S.; You, Y.; Xiao, Y. Research on the Application of Blockchain Smart Contract in Software Supply Chain Management. In *Proceedings of the 2022 3rd Asia Conference on Computers and Communications (ACCC)*, Shanghai, China, 29–31 July 2022; pp. 78–83.
11. MacDonald, N.; Feiman, J. *Magic Quadrant for Application Security Testing*; Gartner Research Report No. G00246914; Gartner, Inc.: Stamford, CT, USA, 2013; pp. 1–29.
12. Setiawan, H.; Erlangga, L.E.; Baskoro, I. Vulnerability Analysis Using The Interactive Application Security Testing (IAST) Approach For Government X Website Applications. In *Proceedings of the 2020 3rd International Conference on Information and Communications Technology (ICOIACT)*, Yogyakarta, Indonesia, 24–25 November 2020; pp. 471–475.
13. Alqaradaghi, M.; Nazir, M.Z.I.; Kozsik, T. Design and Implement an Accurate Automated Static Analysis Checker to Detect Insecure Use of SecurityManager. *Computers* **2023**, *12*, 247. [\[CrossRef\]](#)
14. Pan, Y. Interactive application security testing. In *Proceedings of the 2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, Xiangtan, China, 10–11 August 2019; pp. 558–561.
15. Seth, A.; Bhattacharya, S.; Elder, S.; Zahan, N.; Williams, L. Comparing Effectiveness and Efficiency of Interactive Application Security Testing (IAST) and Runtime Application Self-Protection (RASP) Tools in a Large Java-based System. *arXiv* **2023**, arXiv:2312.17726.
16. Bell, J.; Kaiser, G. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, Portland, OR, USA, 20–24 October 2014; pp. 83–101.

17. Yapu, Z.; Ao, G.; Mingmin, G.; Boyang, G.; Kai, X. Research on Intelligent Supply Chain Risk Assessment System Based on Computer Supply Chain Risk Prevention and Control Technology. In Proceedings of the 2024 Asia-Pacific Conference on Software Engineering, Social Network Analysis and Intelligent Computing (SSAIC), New Delhi, India, 10–12 January 2024; p. 345.
18. Lauinger, T.; Chaabane, A.; Arshad, S.; Robertson, W.; Wilson, C.; Kirda, E. Thou shalt not depend on me: A look at javascript libraries in the wild. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 28 February–1 March 2017.
19. Hu, X.; Jiang, W. Research on network host risk assessment method based on environmental risk level attack graph. In Proceedings of the 2022 International Conference on Informatics, Networking and Computing (ICINC), Nanjing, China, 14–16 October 2022; pp. 159–162.
20. Soto-Valero, C.; Durán, F.; Baudry, B. Bloatware in Java: Analysis and automated removal. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2491–2507.
21. Yitagesu, S.; Zhang, X.; Feng, Z.; Li, X.; Xing, Z. Automatic Part-of-Speech Tagging for Security Vulnerability Descriptions. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021.
22. Ladisa, P.; Plate, H.; Martinez, M.; Barais, O. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–25 May 2023; pp. 1509–1526.
23. Xmirror Security. *Software Supply Chain Security White Paper (2021)*; China Academy of Information and Communications Technology: Beijing, China, 2025.
24. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre Attacks: Exploiting Speculative Execution. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2019; pp. 1–19.
25. Safi, M.; Dadkhah, S.; Shoeleh, F.; Mahdikhani, H.; Molyneaux, H.; Ghorbani, A.A. A Survey on IoT Profiling, Fingerprinting, and Identification. *ACM Trans. Internet Things* **2022**, *3*, 1–39. [CrossRef]
26. Hejderup, J.; van Deursen, A.; Gousios, G. Software ecosystem call graph for dependency management. In Proceedings of the 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 101–104.
27. Chen, K.; Zhao, Y.; Guo, J.; Gu, Z.; Han, L. Accelerating firmware vulnerability detection through directed reaching definition analysis. *ICT Express* **2025**, *11*, 951–956. [CrossRef]
28. Kim, S.; Kim, R.; Park, Y.B. Software Vulnerability Detection Methodology Combined with Static and Dynamic Analysis. *Wirel. Pers. Commun.* **2016**, *89*, 777–793.
29. Imtiaz, N.; Gumussoy, B.A.; Williams, L. Comparison of software composition analysis tools for vulnerability detection. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 336–347.
30. Perez, D.; Livshits, B. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In Proceedings of the 30th USENIX Security Symposium, Online, 11–13 August 2021; pp. 1325–1342.
31. Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; Dolan-Gavitt, B. Examining zero-shot vulnerability repair with large language models. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–25 May 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 2339–2356.
32. Deng, G.; Liu, Y.; Mayoral-Vilches, V.; Liu, P.; Li, Y.; Xu, Y.; Zhang, T.; Liu, Y.; Pinzger, M.; Rass, S. PentestGPT: An LLM-empowered automatic penetration testing tool. *arXiv* **2023**, arXiv:2308.06782.
33. GitHub. Gather and Update All Available and Newest CVEs with Their PoC. Available online: <https://github.com/trickest/cve> (accessed on 29 June 2026).
34. Zenodo. Goblin: Neo4J Maven Central dependency graph. Available online: <https://zenodo.org/records/15481588> (accessed on 30 August 2024).
35. Wang, J.M.; Yang, K. Research on Network Intrusion Detection Based on CNN-BiGRU. *Inf. Rec.-Ing. Mater.* **2024**, *25*, 180–183.
36. Schober, P.; Boer, C.; Schwarte, L.A. Correlation Coefficients: Appropriate Use and Interpretation. *Anesth. Analg.* **2018**, *126*, 1763–1768. [CrossRef] [PubMed]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.