VAWKUM Transactions on Computer Sciences
http://vfast.org/journals/index.php/VTCS@ 2026, ISSN(e): 2308-8168, ISSN(p): 2411-6335
Volume 14, Number 1, January-June 2026 pp:96-109

# Security Vulnerabilities of Eclipse Application Programming Interfaces: An Empirical Analysis

**Simon Kawuma** [1*], **David Sabiiti Bamutura** [2], **Aggrey Obbo** [1], **Moreen Kabarungi** [1], **Kalungi Dickson** [1], **Vicent Mabirizi** [3]

[1] Department of Software and Informatics Engineering, Mbarara University of Science and Technology, Ugandan; [2] Department of Computer Science, Mbarara University of Science and Technology, Ugandan; [3] Department of Computer Science, Kabale University, Ugandan

***Keywords***: *Eclipse, public APIs, internal APIs, security vulnerabilities, Software Quality, Evolution.*

***Journal Info***:
*Submitted: July 19, 2025*
*Accepted: March 28, 2026*
*Published: April 19, 2026*

## ABSTRACT

The Eclipse framework distinguishes between stable public APIs and less stable internal APIs; however, there is no formal assurance that these interfaces have been comprehensively evaluated for security vulnerabilities. Furthermore, previous studies revealed that many developers are unaware of most vulnerabilities. Applications that use security vulnerable APIs risk failing and are prone to malicious attacks if vulnerabilities are not fixed. Security vulnerability discovery is a difficult process and takes at least 3 years to discover and fix them. As a result, developers who depend on affected APIs are frequently compelled either to address these vulnerabilities independently or to discontinue their use. This study aims to identify interfaces within the Eclipse framework that are free from known security vulnerabilities and to recommend them for safer adoption by application developers. To achieve this objective, an empirical investigation was conducted using the SonarQube static analysis tool across 28 major Eclipse releases. The analysis focused on determining the presence and distribution of vulnerability-free interfaces. The study resulted in a dataset comprising approximately 222K public APIs and 292K internal APIs that were found to be free of detected security issues. The findings reveal that, on average, about 91.4% of public APIs and 85.3% of internal APIs across the analyzed releases do not exhibit security vulnerabilities. Furthermore, the average time required to remediate identified vulnerabilities was estimated at 1,425 days. These results offer valuable guidance for both API providers and users by identifying more reliable interfaces and providing an indication of the effort required to resolve security issues. Additionally, the dataset of vulnerability-free interfaces has been made publicly accessible via GitHub to support the development of more secure software systems.

***Correspondence author email address:** simon.kawuma@must.ac.ug
DOI: 10.21015/vtcs.v14i1.2196

## 1 Introduction

Software systems are often developed by leveraging existing frameworks and libraries. This approach enhances productivity while promoting the reuse of established functionality [1, 2]. Consequently, widely adopted application frameworks, including Eclipse [3], MSDN [4], jBPM [5], and JUnit [6], typically expose public (stable) interfaces (APIs) to support developers in building their

This work is licensed under a Creative Commons Attribution 3.0 License.

applications.

Alongside public APIs, these frameworks also expose internal interfaces. A prominent and widely adopted example is the Eclipse application framework. Eclipse is a large-scale, complex open-source system that is utilized by thousands of developers. Over a period exceeding two decades, it has continuously evolved, resulting in more than 28 major releases and 55 minor updates. Frameworks such as Eclipse, jBPM, and JUnit follow a naming convention for internal interfaces by including the sub-string `internal` within their package structures, whereas in the JDK, internal APIs are typically identified by package names that begin with the sub-string `sun`.

Framework developers typically advocate for the use of public APIs since they are viewed as reliable, well-supported, and sufficiently mature. In contrast, internal APIs are generally discouraged because they tend to lack stability, formal support, and maturity, and may be altered or removed without prior warning [3, 4, 6, 7]. Nevertheless, the use of internal APIs remains prevalent in practice. Empirical findings by Businge et al. [8, 9] indicate that approximately 44% of 512 examined Eclipse plug-ins rely on internal APIs. Likewise, Hora et al. [10] reported that 23.5% of 9,702 Eclipse client projects hosted on GitHub make use of internal interfaces. Furthermore, experienced developers have noted that utilizing internal APIs can, in some cases, be more advantageous than implementing equivalent functionality independently from the ground up [11].

Developers inevitably rely on both public and internal APIs, as their use accelerates development and shortens the time required to deliver applications to market. While interface providers generally assert that public APIs are officially supported and internal APIs are not, there is no assurance that either category has undergone thorough security testing for software issues like bugs, technical debt, security vulnerabilities etc. Furthermore, many developers are unaware of most vulnerabilities [12]. Applications that use security vulnerability interfaces risk failing and are prone to malicious attacks and security failures if the security vulnerabilities are not fixed [13]. This suggests that application developers should be prepared to address security vulnerabilities independently when they arise. Alternatively, remediation may also be undertaken by framework maintainers on behalf of the users. however, security vulnerabilities discovery is difficulty and it takes at least 3 years to discover and fix them [14], thus interface users might wait indefinitely for vulnerability fixation by provider.

Many applications depend on vulnerable packages thus it is critical to discover and fix vulnerabilities [14]. This indicates that interface users are often compelled either to resolve the vulnerabilities on their own or to discontinue using the affected interfaces. To avoid prolonged dependence on interface providers for fixes, or the burden of handling vulnerability remediation, developers could instead rely on interfaces that are free from known security issues. However, in practice, many users may lack awareness of such vulnerability-free interfaces within the Eclipse framework.

Furthermore, developers often perform manual searches to locate the required functionality within the Eclipse framework [11]. Given the size and complexity of Eclipse, there is a high likelihood that users may initially encounter interfaces containing security vulnerabilities rather than those that are free from such issues when identifying components for use in their applications. Using a static code analysis tool, namely SonarQube [15], we carried out an investigation on 28 major Eclipse frameworks releases with a goal to establish: 1) composition of security vulnerabilities in 28 major Eclipse releases and 2) the presence of security vulnerability-free interfaces throughout the evolution of the Eclipse framework. The objective of the study is to recommend the identified secure interfaces to application developers. Guided by these objectives, the following research questions were formulated as follows:

***RQ1: What is the distribution and composition of security vulnerabilities across Eclipse framework releases?*** Understanding the overall number of security vulnerabilities, their distribution, the most frequently occurring types, and the effort required for remediation is essential for enhancing the quality of the Eclipse framework and its interfaces. Such insights enable both framework developers and interface users to better assess and estimate the effort needed to address security

vulnerabilities within a specific Eclipse release.

***RQ2: Is it possible to identify Eclipse framework interfaces that are free from security vulnerabilities?*** Addressing and resolving security vulnerabilities often requires a considerable amount of time, which may force Eclipse interface users to wait for extended periods before fixes are provided by framework developers. In this study, we conducted an empirical analysis to determine whether security vulnerability-free interfaces exist within the Eclipse framework. The goal was also to raise awareness among developers about such secure interfaces, as they may otherwise unknowingly rely on vulnerable ones, thereby exposing their applications to potential threats. To answer thees research questions, we employed the static analysis tool SonarQube [15] to collect data on security vulnerabilities. A comprehensive description of the research methodology adopted in this study is presented in Section 4.

Although a number of studies have been conducted about vulnerability detection and analysis, no study has been carried out to ascertain the existence of vulnerability-free interfaces in Eclipse frameworks, therefore, The contributions of this study can be summarized in three main aspects:

1. We present a dataset comprising of 222K public API classes and 292K internal API classes that are free from identified security vulnerabilities. This dataset can be utilized by both Eclipse interface providers and application developers. Interface providers may leverage it to estimate the effort required for vulnerability remediation, while developers can consult it to identify secure interfaces suitable for use in their applications.
2. Although Eclipse interface providers assert that public APIs are stable and reliable [3], our empirical findings support this claim by showing that more than 91.4% of public APIs across all analyzed Eclipse releases are free from security vulnerabilities.
3. While the use of internal APIs is generally discouraged due to concerns regarding instability and lack of support [3, 4, 6], our results indicate that a significant proportion of these interfaces are not inherently problematic. Specifically, over 85.3% of internal APIs were found to be free from security vulnerabilities across the studied releases, suggesting that they can also be considered for use by developers.

The structure of this paper is as follows: Section 2 provides background information on Eclipse interfaces, security vulnerabilities, and the SonarQube tool employed in this study. Section 3 reviews existing related work. The research methodology is detailed in Section 4. Section 5 presents the results and key findings of the study. A discussion of these findings is given in Section 6. Threats to the validity of the study are examined in Section 7. Finally, Section 8 summarizes the conclusions and highlights directions for future research.

## 2 Background

This section provides the essential background on the different types of interfaces in the Eclipse framework, outlines key concepts related to security vulnerabilities, and describes the static analysis tools used to examine defects within the framework throughout its evolution.ts within the framework throughout its evolution.

### 2.1 Eclipse internal APIs

These elements represent internal implementation artifacts which, following the Eclipse naming convention [3], are located in packages whose fully qualified names contain the sub-string `internal`. Such artifacts may consist of public Java classes or interfaces, as well as public or protected methods and fields defined within them. The use of internal APIs is generally discouraged due to their potential instability [16]. Eclipse explicitly warns that developers who choose to rely on these interfaces do so at their own risk, as internal APIs may be modified or removed at any time without prior notice. In addition, these APIs are typically not accompanied by official documentation or support.

### 2.2 Eclipse Public APIs

These elements consist of public Java classes or interfaces located in packages whose fully qualified names do not include the segment `internal`, as well as public or protected methods and fields defined within such classes or interfaces. According to Eclipse, public APIs are regarded as stable and are therefore considered

safe for use by application developers. In addition, Eclipse offers official documentation and support for these public interfaces.

### 2.3 Security Vulnerabilities

Security vulnerabilities are weaknesses within a system that can be exploited by malicious actors, such as attackers, to carry out unauthorized actions in a computer system. Vulnerability is a weakness in computer systems [17]. It is a possible area that is liable to being used for attacks. A security vulnerability is a weakness within a system that can be exploited by an attacker. Additionally, a security vulnerability may be described as a defect such as a bug, flaw, error, fault, gap, or weakness present within the software architecture, design, code, or implementation, which can be exploited by malicious actors. In this study, we thoroughly scrutinized security vulnerabilities in puplic APIs and internal APIs of the Eclipse framework as these are the interfaces containing classes that software developers use when developing their application.

### 2.4 SonarQube

In this study, we employed SonarQube, a widely recognized open-source static code analysis tool that is extensively used in both academic research [18] and industrial practice [19]. SonarQube is available either as a cloud-based service platform[1] or as a downloadable solution that can be deployed on a local server. The tool computes a range of metrics, including lines of code, code complexity, and adherence to predefined coding standards across commonly used programming languages [20]. Whenever the analyzed code violates a specified rule or exceeds a defined threshold for a given metric, SonarQube reports an issue. It provides rule sets that address reliability, maintainability, and security concerns. Additionally, SonarQube supports multiple programming languages such as Java, Python, C++, and JavaScript, each with its own set of rules. For this research, we utilized SonarQube version 8.2, which includes more than 718 rules specifically for Java. The full catalog of these rules is publicly accessible online[2].

[1] sonarcloud.io
[2] https://rules.sonarsource.com/java

## 3 Related Work

This section outlines how the present study relates to prior research. Earlier work by Businge et al. [8, 9, 11, 21–23] focused on empirical investigations of the co-evolution between the Eclipse SDK framework and its third-party plug-ins (ETPs). In these studies, the authors examined how changes in Eclipse interfaces utilized by ETPs influenced their compatibility with subsequent framework releases. Their analyses were based on source code and considered only open-source ETPs.

In a later study, Businge et al. [11] extended this work by incorporating survey data, thereby including commercial ETPs and addressing human-related factors. A key finding across these studies is that interface users frequently rely on unstable interfaces, largely due to the absence of stable alternatives that provide equivalent functionality. Supporting this observation, Kawuma et al. demonstrated that fewer than 1% of APIs offer functionality comparable to that of non-APIs [2]. While the earlier studies by Businge et al. examined both public and internal APIs, they did not investigate the identification of security vulnerability-free interfaces, which is the primary focus of the present study.

In a recent study, Businge et al. [1] employed a clone detection technique to examine the stability of internal interfaces as the Eclipse framework evolves. Their findings identified approximately 327K internal interfaces that remained stable, which they proposed as potential candidates for promotion. Closely related to this work, Hora et al. [10] investigated the migration of interfaces from internal to public status. Their results showed that about 7% of 2,277 internal interfaces were eventually promoted to public APIs, and these promoted interfaces tended to attract a larger number of clients. In a similar line of research, Kawuma et al. [24] reported that the transition of non-APIs to APIs occurs at a slow rate and typically requires a considerable amount of time. A similar study by Kawuma et. al [25] looked at existence of APIs that have no bugs in Eclipse and they discovered that there exist 217K and 302K public and internal APIs without bugs respectively in all the studied Eclipse releases. Waqas et.al [26] conducts a comparative examination of several machine-learning techniques to tackle the bug localization difficulty, they

dicovered that Adaptive Bug Localization System (ABLS) model indicated a superior performance.

Moreover, a recent empirical study by Kawuma et al. [27] applied SonarQube analysis across 28 major Eclipse releases and identified approximately 218K public APIs and 321K internal APIs that were free from code smells. The study further reported that, across all examined releases, 87.3% of the interfaces corresponded to public APIs, while 91.5% were internal APIs. Although prior work in [1], [10], [27], [25], and [24] has focused on identifying and recommending internal interfaces for promotion, none of these studies examined the presence of security vulnerability-free interfaces within the Eclipse framework.

Several machine learning, deep learning techniques and tools,have been proposed in previous work to detect security vulnerabilities[28–32]. For instance, Zhang et al. introduced SecureCodeBERT [33], a transformer-based model specifically designed to identify and classify high-risk security vulnerabilities in PHP applications. The model achieves a precision of 0.892 and a recall of 0.867, demonstrating notable improvements compared to conventional static analysis approaches. Furthermore, Large language models (LLMs) have been used as transformative tools in the detection and management of software vulnerabilities and these LLMs models have proved themselves superior to traditional tools [34]. Although some of the above studies look at security vulnerability detection tools, ML and DL learning techniques, none of the above studies examined aspects such as the composition and distribution of security vulnerabilities, the effort required for their remediation, as well as the presence of vulnerability-free interfaces within the Eclipse framework, in contrast to the focus of our study.

# 4 Research Methodology

This section describes the experimental design and outlines the procedures followed to collect data used in addressing the research questions.

## 4.1 Eclipse Releases Collection

In this section, we describe the data sources used in the study. The analysis is based on 28 major Eclipse SDK releases obtained from the Eclipse Project Archive website [35, 36]. Table 1 summarizes the selected releases. The first column lists the major versions, ranging from Eclipse–1.0 (E-1.0) to Eclipse–4.16 (E-4.16), while the second column indicates their respective release dates. The third column reports the Java lines of code (LOC) for each release, and the fourth column presents the total number of Java classes contained in each version.

Eclipse was chosen as the subject of this research due to its widespread adoption as an open-source framework, making it highly relevant to a large developer community. The framework undergoes continuous evolution, with new versions typically released every three months. This regular release cycle provides an opportunity to analyze trends in security vulnerabilities over time. The study specifically focuses on major releases because transitions between these versions often involve the addition, modification, or removal of projects, sub-projects, packages, classes, interfaces, fields, and methods within the framework.

## 4.2 Security Vulnerabilities Extraction

In this section, we describe the process used to collect data for addressing research questions **RQ1** and **RQ2**. The SonarQube tool (version 8.2) [15] was employed to obtain information on security vulnerabilities across the different Eclipse releases. This tool was selected due to its widespread adoption in both academic research [18, 37] and industrial practice [19, 38]. We configured and executed SonarQube locally, enabling all 58 Java security-related rules designed for vulnerability detection. A violation of any of these rules indicates the presence of a security vulnerability in the analyzed source code.

Our analysis focused on determining the total number of security vulnerabilities, estimating the effort required for their remediation, and identifying the most prevalent vulnerabilities defined as the most frequently violated rules for each Eclipse major release. In addition to detecting vulnerabilities, SonarQube provides an estimate of remediation effort expressed in days, assuming a standard working duration of eight hours per day [15].

Figure 1 depicts the procedure adopted to identify and extract security vulnerability information across all analyzed Eclipse releases. SonarQube processes source directories containing Java files as input and analyzes them to detect potential security vulnerabilities

**Table 1.** Eclipse major releases and their corresponding release dates

| Major Releases | Release Date | Java LOC | Java Classes | Major Release | Release Date | Java LOC | Java. Classes |
|---|---|---|---|---|---|---|---|
| E-1.0 | 07-Nov-01 | 449K | 4,608 | E-4.2 | 27-Jun-12 | 2.8M | 22,443 |
| E-2.0 | 27-Jun-02 | 769K | 6,751 | E-4.3 | 05-Jun-13 | 2.9M | 22,798 |
| E-2.1 | 27-Mar-03 | 959K | 7,911 | E-4.4 | 06-Jun-14 | 3.1M | 23,880 |
| E-3.0 | 25-Jun-04 | 1.3M | 10,634 | E-4.5 | 03-Jun-15 | 3.14M | 23,920 |
| E-3.1 | 27-Jun-05 | 1.6M | 12,299 | E-4.6 | 06-Jun-16 | 3.2M | 23,936 |
| E-3.2 | 29-Jun-06 | 2M | 14,941 | E-4.7 | 28-Jun-17 | 3.3M | 25900 |
| E-3.3 | 25-Jun-07 | 2.1M | 16,036 | E-4.8 | 27-Jun-18 | 3.39M | 26,180 |
| E-3.4 | 17-Jun-08 | 2.5M | 18,800 | E-4.9 | 19-Sept-18 | 3.4M | 26,363 |
| E-3.5 | 11-Jun-09 | 2.6M | 19,169 | E-4.11 | Mar-19 | 3.5M | 27,448 |
| E-3.6 | 08-Jun-10 | 2.7M | 20,922 | E-4.12 | Jun-19 | 3.51M | 27,784 |
| E-3.7 | 13-Jun-11 | 2.75M | 21,104 | E-4.13 | Sept-19 | 3.52M | 27,904 |
| E-3.8 | 27-Jun-12 | 2.8M | 22,477 | E-4.14 | Dec-19 | 3.53M | 27,976 |
| E-4.0 | 27-Jul-10 | 2.6M | 20,498 | E-4.15 | Mar-20 | 3.55M | 28,500 |
| E-4.1 | 20-Jun-11 | 2.7M | 21,234 | E-4.16 | Jun-20 | 3.6M | 28,135 |

within specific classes. For each Eclipse release, the tool generates a report that can be accessed through the SonarQube server at the URL: http://localhost:9000 . An example of such a report for Eclipse-4.16 is presented in Figure 1.

Within these reports, each file is assigned a security rating (SR) by SonarQube based on both the type and number of vulnerabilities identified in the corresponding class. For instance, as illustrated in Figure 1, files in the last row are assigned a rating of A, indicating the absence of security vulnerabilities. The tool also records the number of vulnerabilities detected in each class. In addition, SonarQube categorizes file security ratings as follows: A denotes no vulnerabilities, B indicates at least one minor vulnerability, C represents at least one major vulnerability, D corresponds to at least one critical vulnerability, and E signifies the presence of at least one blocker-level vulnerability.

Furthermore, SonarQube aggregates the total number of vulnerabilities identified in each release and estimates the effort required for their remediation. For example, as shown in Figure 1, a total of 2K vulnerabilities were detected, requiring approximately 10 days to address. In this study, we considered the total number of vulnerabilities reported per class for each Eclipse release. Particular attention was given to classes with a rating of A, indicating no detected vulnerabilities. To compute the proportion of vulnerability-free classes, we calculated the ratio of classes with rating A to the total number of classes within each Eclipse release.

## 5 Results

In this section, we report and analyze the data obtained in Section 4 to address research questions RQ1 and RQ2.

### 5.1 Security Vulnerabilities and Remediation Efforts

Figure 2 illustrates the results related to the total number of security vulnerabilities and the corresponding remediation effort required to address them across the analyzed Eclipse releases. In this figure, the bar chart depicts the total count of security vulnerabilities, whereas the line chart shows the estimated remediation effort in days. Considering both visualizations, a steady upward trend is observed in both the number of vulnerabilities and the effort required for their resolution. This pattern can be explained by the continuous evolution of the Eclipse framework, where the introduction of new functionalities such as additional projects, classes, methods, and increased lines of code contributes to the emergence of new security vulnerabilities.

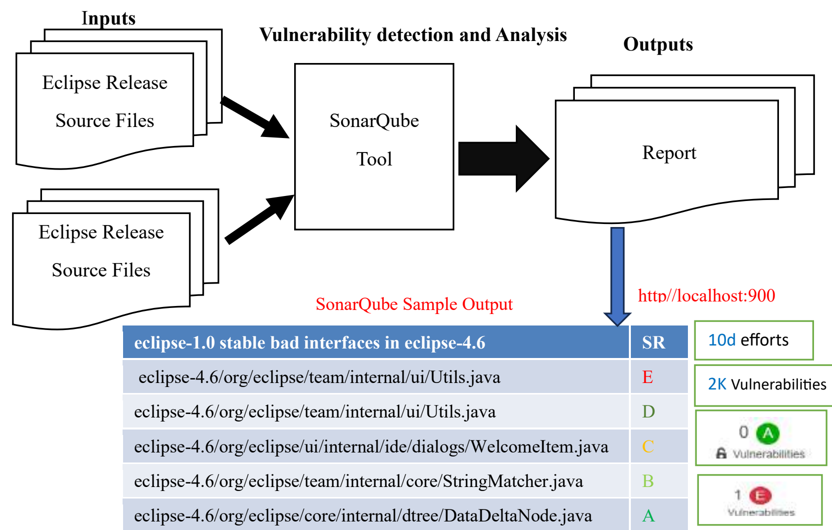


**Figure 1.** SonarQube vulnerabity detection tool

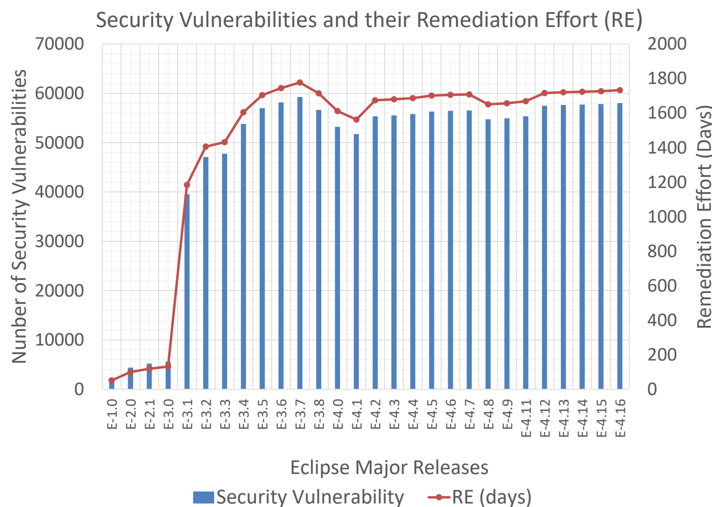


**Figure 2.** Total number of security vulnerability and remediation effort in Eclipse releases

A decrease in both the number of security vulnerabilities and the corresponding remediation effort is observed between Eclipse–3.8 and Eclipse–4.0, followed by a steady increase from Eclipse–4.1 onwards. This temporary decline between Eclipse–3.8 and Eclipse–4.0 can be explained by the removal of 1,979 classes from Eclipse–3.8, as indicated in Table 1. The elimination of these classes likely resulted in the removal of associated vulnerabilities, thereby reducing the effort required for remediation.

Additionally, this shift can be linked to architectural changes introduced by Eclipse developers. An evaluation of the Eclipse 3.x architecture revealed limitations in accommodating emerging technologies, supporting community growth, and attracting new contributors. Consequently, the architecture was redesigned starting from version 4.0 to address these challenges [2].

From the bar chart in Figure 2, it is evident that the total number of security vulnerabilities across the analyzed releases ranges from 2,227 to 59,244. Furthermore, the estimated remediation effort varies between 51 and 1,777 days for addressing vulnerabilities in the examined Eclipse versions.

## 5.2 Security vulnerability-free interfaces in Eclipse releases

Figure 5 shows the results related to the proportion of security vulnerability-free interfaces across different Eclipse major releases. In this figure, the first and second bars represent the percentages of security vulnerability-free public APIs and non-APIs (internal APIs), respectively, relative to their total counts within each category. Further examination of Figures 3 and 4 indicates that more than 91.4% of public APIs and 85.3% of non-APIs are free from security vulnerabilities. Additionally, in Figure 5, the bar charts illustrate the proportion of vulnerability-free public API and non-API

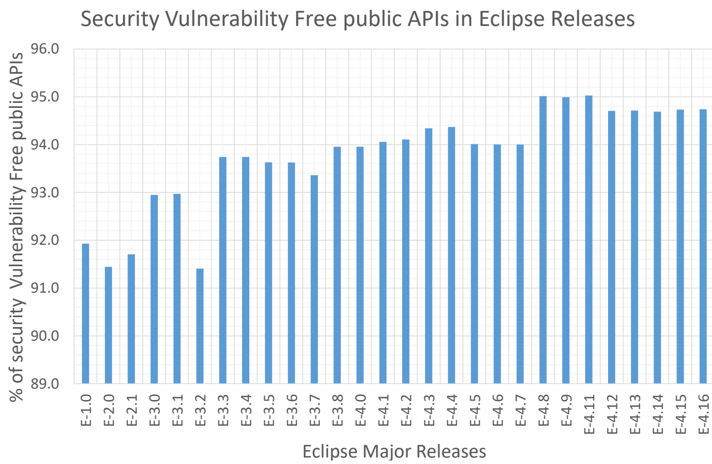


**Figure 3.** Security vulnerability free puplic APIs

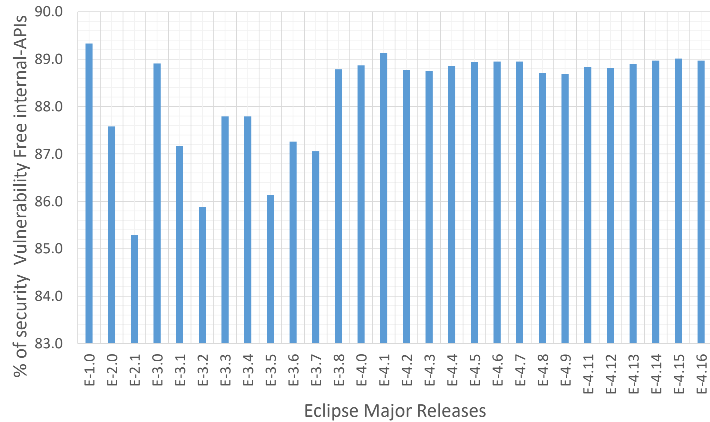


**Figure 4.** Security vulnerability free non-APIs

classes relative to the total number of interfaces per release, while the line chart represents the overall count of vulnerability-free interfaces, obtained by summing both categories.

For each Eclipse release in Figure 5, two bars are displayed: the first corresponds to the percentage of vulnerability-free public API classes, and the second represents vulnerability-free internal API (non-API) classes. Observing the bar charts, it is evident that a larger share of vulnerability-free classes belongs to the internal API category compared to public APIs. This trend is influenced by the fact that internal APIs are approximately twice as numerous as public APIs throughout the evolution of Eclipse [2]. Across all analyzed releases, the proportion of vulnerability-free public API classes

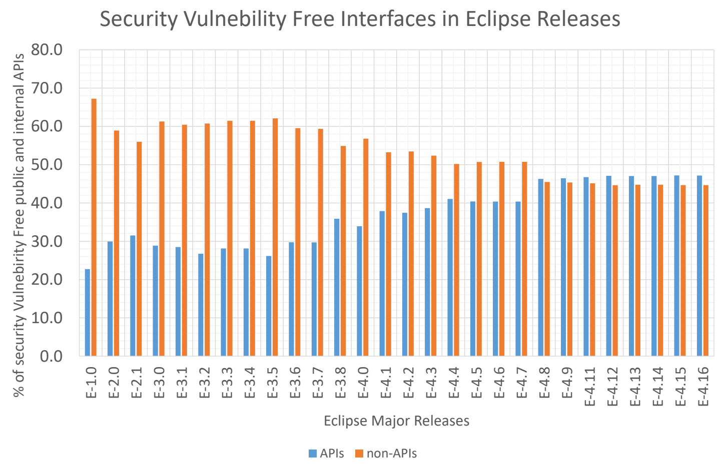


**Figure 5.** Security vulnerability free APIs and non-APIs

ranges between 22.7% and 47.2% of the total classes, whereas internal API classes range from 44.7% to 67.2%. On average, vulnerability-free public APIs and internal APIs account for 36.8% and 53.6% of the total classes in a given release, respectively.

Although internal APIs (non-APIs) are generally regarded as unstable, unsupported, and subject to modification or removal during framework evolution [3, 4, 6], one might expect a higher prevalence of vulnerabilities within this category. However, the findings of this study indicate otherwise, revealing that, on average, 53.6% of all classes across the examined Eclipse releases exhibit no security vulnerabilities.

## 5.3 Common security vulnerabilities in Eclipse Releases

In this section, we present results of the common security vulnerabilities found in Eclipse releases. In this research, we considered 58 security vulnerability rules provided in SonarQube tool (version 8.2) to detect security vulnerabilities in Eclipse releases. The complete list of security vulnerability rules is available online[3]. security vulnerability rules create code violations that represents something wrong in the code which will be reflected as a security vulnerability. Tables 3-6 presents results of common security vulnerabilities that arise as a result of violation of standard security vulnerability rules in the SonarQube. The first column in the tables show the unique

[3] https://rules.sonarsource.com/java

rule id whereas the second column show a brief description of the rule. The third column (vulnerability) shows the total number of security vulnerabilities that are generated as a consequence of violating a given security vulnerability rule in all the analyzed Eclipse releases.

## 6 Discussion

From Figure 2, a generally increasing trend in the number of security vulnerabilities is observed across the analyzed Eclipse major releases. This pattern can be explained by the continuous evolution of the Eclipse framework, where additional functionality—such as new projects, classes, and methods—is introduced, leading to an increase in lines of code (LOC). As a result, the expansion of functionality is often accompanied by the introduction of new security vulnerabilities. Moreover, Eclipse benefits from a large and active community of developers and contributors who continuously add to its extensive codebase [2].

The total number of detected security vulnerabilities also provides an indication of the time and effort required by both framework developers and interface users to address these issues. Additionally, as illustrated in Figures 3 and 4, more than 91.4% of public interfaces and 85.3% of internal interfaces are free from vulnerabilities across all analyzed Eclipse major releases. This observation suggests that a substantial proportion of Eclipse interfaces undergo adequate testing before being integrated into the framework ecosystem.

Tables 2 3, 4, 5 present the most frequently occurring security vulnerabilities identified by SonarQube across the studied Eclipse releases. These findings are valuable for both interface providers and users, as they highlight common vulnerability patterns and emphasize the importance of adhering to sound coding practices to minimize security risks in applications.

The SonarQube static analysis tool was used in this research to address the trends of security vulnerabilities in the Eclipse framework, and the aim was to find out the vulnerability-free interfaces that can be suggested to developers. The choice of static analysis tools was due to the fact that they are free, easily available, and can be used to detect vulnerabilities at an earlier stage of the development cycle, when fixing them is cheaper [39]. These tools generate alerts based on predefined programming rules to detect defects [39]. For example, SonarQube includes 58 security-related rules, and violations of these rules indicate the presence of security vulnerabilities that developers can address.

However, static analysis tools often produce a large volume of warnings, which can be challenging for users to interpret. In this regard, SonarQube provides user-friendly and interactive interfaces that facilitate efficient analysis and interpretation of vulnerability reports.

### 6.1 Comparative Analysis with Other Open-Source Ecosystems

To enhance the generalizability of our findings, we compare the security vulnerability patterns observed in the Eclipse framework with findings from other widely adopted open-source ecosystems, notably Android APIs and OpenJDK libraries.

Previous studies on Android ecosystem (e.g., Mohayeji et al. [40]; Gao et al. [41]) indicate that software dependencies often contain known vulnerabilities, with developers frequently postponing updates to affected libraries. Similarly, empirical analysis of Python and Java ecosystems ( e.g., Kikas et al. [42]; Zimmermann et al. [43] ) reveals that vulnerability prevalence is non-trivial, with many packages exhibiting security weaknesses over time due to dependency complexity and delayed updates.

In contrast, our findings reveals that over 91.4% public APIs and 85.3% of internal APIs in Eclipse are security vulnerability-free across all analysed releases. This percentage is relatively high compared to reports from other ecosystems such as Maven and PyPI, where empirical studies report widespread vulnerability exposure, long resolution time (often several years), and significant propagation through transitive dependencies [44, 45]

Furthermore, studies on OpenJDK and Java libraries have highlighted risks associated with internal or undocumented APIs (e.g., usage of sun.* packages), which are often unstable and may introduce security vulnerabilities due to lack of official support and evolution guarantees [46]. Empirical research has also shown that reliance on internal APIs increases maintenance challenges and potential security risks as these interfaces may change or be removed without notice [47]. However, our results

**Table 2.** Common blocker security vulnerabilities in Eclipse releases

| Rule | Rule Description | Vulnerability |
|---|---|---|
| S2976 | File createTempFile should not be used to create a directory | 140 |
| S2647 | Basic authentication should not be used | 137 |
| S5547 | Cipher algorithms should be robust | 27 |
| S4830 | Server certificates should be verified during SSL/TLS connections | 26 |
| S5527 | Server hostnames should be verified during SSL/TLS connections | 17 |

**Table 3.** Common critical security vulnerabilities in Eclipse releases

| Rule | Rule Description | Vulnerability |
|---|---|---|
| S2755 | - XML parsers should not be vulnerable to XXE attacks | 3501 |
| S5542 | - Encryption algorithms should be used with secure mode | 149 |

**Table 4.** Common minor security vulnerabilities in Eclipse releases

| Rule | Rule Description | Vulnerability |
|---|---|---|
| S3066 | enum fields should not be publicly mutable | 719146 |
| S1444 | public static fields should be constant | 566304 |
| S1104 | Class variable fields should not have public accessibility | 25521 |
| S1989 | Exceptions should not be thrown from servlet methods | 8620 |
| S2386 | Mutable fields should not be public static | 7724 |
| S899 | Return values with operation status code shouldn't be ignored | 17 |
| S1148 | Throwable.printStackTrace(...) should not be called | 9 |

**Table 5.** Common major security vulnerabilities in Eclipse releases

| Rule | Rule Description | Vulnerability |
|---|---|---|
| S4423 | Weak SSL/TLS protocols should not be used | 55 |

suggest that in Eclipse, even internal APIs exhibit a high proportion (85.3%) of vulnerability-free interfaces, challenging the common assumption that internal APIs are inherently less secure.

Another point of comparison lies in remediation effort. Prior research indicates that vulnerability resolution in software ecosystems can take several months to years [45]. Our study aligns with this trend, showing an average remediation effort of 1425 days, reinforcing the general difficulty of vulnerability detection and resolution across ecosystems.

## 7 Threats to Validity

As with any empirical investigation, our study is subject to potential threats to validity. These threats can be divided into construct, internal and external validity.

validity is related to the level at which the metrics adopted are appropriate to measure the phenomena being studied. In this study, the approach used to compute the proportion of security vulnerability-free interfaces in Eclipse may introduce construct validity limitations. This is owing to the fact that only classes are analyzed whereas other components of the program that are relevant like methods and variable declarations are not taken into account.

*Internal validity* relates to factors that may influence the accuracy of the results due to the tools and procedures employed. In our case, the use of SonarQube for data extraction introduces potential internal validity con-

cerns, as different tools might yield different outcomes. Additionally, like most static analysis tools, SonarQube does not achieve perfect accuracy. It identifies vulnerabilities based on predefined rules and known patterns; however, some vulnerabilities may only become apparent during compilation or runtime and therefore may not be detected by static analysis.

*External validity* addresses the extent to which the findings can be generalized beyond the studied context. This research focuses on the Eclipse SDK framework, a widely used and large-scale open-source system, making it a relevant and representative case study. Its open-source nature also ensures accessibility of the source code. Nevertheless, as is typical in empirical software engineering studies, the results cannot be directly generalized to other systems, particularly those developed using programming languages other than Java.

## 8 Conclusion and Future Work

In this study, we conducted an empirical analysis of 28 major Eclipse releases to determine the proportion of vulnerability-free interfaces. The results indicate that more than 91.4

As part of future work, we intend to investigate the popularity of the identified vulnerability-free interfaces by examining both their internal and external usage. Internal usage will be assessed by analyzing how frequently these interfaces are used within Eclipse packages and libraries. External usage will be evaluated by determining the extent to which applications hosted on GitHub depend on these vulnerability-free interfaces.

Additionally, external adoption can be further measured by examining the number of developers who interact with or utilize these interfaces. Although a large proportion of interfaces are free from security vulnerabilities, this alone does not guarantee overall software quality. Therefore, we plan to extend this work by evaluating additional quality attributes of the identified interfaces, including factors such as complexity and documentation.

Also, comparative study can be carried out between SOTA models and LLM/ML models and sonarqube tools to compare the findings of all tools interms of their precision and recall. Furthermore, the dataset in this study can be used as a fundation for the development of an Eclipse plugin that can be used to automate security vulunerability assessment during API evolution.

## Acknowledgment

The authors express their gratitude to the staff of the Computing Services Department at Mbarara University of Science and Technology for providing server space that facilitated the execution of the study experiments. The authors also confirm that no AI tools were utilized in generating the research data, conducting the analysis, producing the results, interpreting the findings, or preparing the cited scholarly content.

## Authors Contribution

**Kawuma Simon:** Conceptualization, Methodology, Formal analysis, Resources, Data curation, conducting experiment, Writing - original draft, Writing - review & editing, Visualization. **David Sabiiti Bamutura:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing, Visualization. **Aggrey Obbo:** performing experiments, or data/evidence collection, Writing - original draft, Writing - review & editing. **Moreen Kabarungi:** Writing - original draft, Writing - review & editing. **Kalungi Dickson:** Writing - original draft, Writing - review & editing. **Mabirizi Vicent:** Writing - original draft, Writing - review & editing, performing the experiments, or data/evidence collection.

## Funding Information

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sector.

## Data availability

To allow independent replication and verification of the study, we provide a full replication package on Github with *the detailed lists of both bug-free public and internal APIs in different Eclipse releases*. [4]

## Declarations

The authors declare no conflict of interest.

[4] https://github.com/simonkawuma/Security-Vulnerability--free-Eclipse-Interfaces

## References


[1] J. Businge, S. Kawuma, M. Openja, E. Bainomugisha, and A. Serebrenik, "How stable are eclipse application framework internal interfaces?" in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 117–127.

[2] S. Kawuma, J. Businge, and E. Bainomugisha, "Can we find stable alternatives for unstable eclipse interfaces?" in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[3] J. des Rivières, "How to use the Eclipse API," http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html, consulted January, 2020.

[4] Oracle, "Why developers should not write programs that call 'sun' packages," https://www.oracle.com/technetwork/java/faq-sun-packages-142232.html, consulted October, 2020.

[5] T. jBPM Team, "The jbpm api," http://docs.jboss.org/jbpm/v5.0/userguide/ch05.html#d0e2099, consulted October, 2020.

[6] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, and C. Stein, "Junit 5 user guide," https://junit.org/junit5/docs/current/user-guide/#api-evolution, consulted October, 2020.

[7] E. Foundation, "Evolving Java-based APIs," https://wiki.eclipse.org/Provisional_API_Guidelines, consulted July, 2020.

[8] J. Businge, A. Serebrenik, and M. G. Van Den Brand, "Eclipse api usage: the good and the bad," *Software Quality Journal*, vol. 23, no. 1, pp. 107–141, 2015.

[9] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "Eclipse API usage: the good and the bad," in *SQM*, 2012, pp. 54–62.

[10] A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 278–289.

[11] J. Businge, A. Serebrenik, and M. G. J. v. Brand, "Analyzing the Eclipse API usage: Putting the developer in the loop," in *17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, 2013, pp. 37–46.

[12] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[13] P. Wang, S. Liu, A. Liu, and W. Jiang, "Detecting security vulnerabilities with vulnerability nets," *Journal of Systems and Software*, vol. 208, p. 111902, 2024.

[14] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[15] A. Campbell, "Metric definitions - sonarqube documentation," https://docs.sonarqube.org/display/SONAR/Metric+Definitions, consulted July, 2021.

[16] J. des Rivières, "Evolving Java-based APIs," http://wiki.eclipse.org/Evolving_Java-based_APIs, consulted January, 2020.

[17] P. Paul, A. Bhuimali, P. Aithal, and R. Rajesh, "Vulnerability in information technology and computing—a study in technological information assurance," *International Journal of Management, Technology, and Social Sciences (IJMTS)*, vol. 4, no. 2, pp. 87–94, 2019.

[18] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *International Conference in Software Engineering for Defence Applications*. Springer, 2018, pp. 165–175.

[19] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, 2020.

[20] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 501–511.

[21] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "Survival of Eclipse third-party plug-ins," in *ICSM*, 2012, pp. 368–377.

[22] ——, "An empirical study of the evolution of Eclipse third-party plug-ins," in *EVOL-IWPSE'10*. ACM, 2010, pp. 63–72.

[23] J. Businge, A. Serebrenik, and M. G. J. v. Brand, "Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases," in *12th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 164–173.

[24] S. Kawuma and E. Nabaasa, "Identification of promoted eclipse unstable interfaces using clone detection technique," *International Journal of Software Engineering and Application*, 2018.

[25] S. Kawuma, D. S. Bamutura, A. Obbo, V. Mabirizi, M. Kabarungi, and E. Nabaasa, "Eclipse application programming interfaces: How buggy are they?" *VFAST Transactions on Software Engineering*, vol. 13, no. 2, pp. 228–244, 2025.

[26] W. Ali, S. Lakho, N. N. Bhatti, I. A. Memon *et al.*, "Adaptive bug localization framework for precision-driven bug localization in software engineering," *VFAST Transactions on Software Engineering*, vol. 12, no. 3, pp. 230–242, 2024.

[27] S. Kawuma, D. S. Bamutura, A. Obbo, and E. Nabaasa, "Investigation of code smells in eclipse framework using sonarqube: An empirical analysis," *International Journal of Software Engineering and Computer Systems*, vol. 11, no. 2, pp. 176–187, 2025.

[28] H. Arif, A. K. S. Ali, and H. A. Nabi, "Iot security through ml/dl: Software engineering challenges and directions," *ICCK Journal of Software Engineering*, vol. 1, no. 2, pp. 90–108, 2025.

[29] I. Rakine, A. Oukaira, K. El Guemmat, I. Atouf, S. Ouahabi, M. Talea, and T. Bouragba, "Comprehensive review of intrusion detection techniques: Ml and dl in different networks," *IEEE Access*, 2025.

[30] R. Batool, A. Naseer, A. Maqbool, and M. Kayani, "Automated categorization of software security requirements: an nlp and ml based approach," *Requirements Engineering*, pp. 1–13, 2025.

[31] N. Shiri Harzevili, A. Boaye Belle, J. Wang, S. Wang, Z. M. Jiang, and N. Nagappan, "A systematic literature review on automated software vulnerability detection using machine learning," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–36, 2024.

[32] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, "Characterizing and understanding software security vulnerabilities in machine learning libraries," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 27–38.

[33] J. Zhang, "Securecodebert: An ai-powered model for identifying and categorizing high-risk security vulnerabilities in php-based critical infrastructure applications," *Journal of Sustainability, Policy, and Practice*, vol. 1, no. 4, pp. 80–94, 2025.

[34] S. M. Taghavi Far and F. Feyzi, "Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics," *International Journal of Information Security*, vol. 24, no. 2, p. 78, 2025.

[35] E. Foundation, "Eclipse project archived download," http://archive.eclipse.org/eclipse/downloads/index.php, consulted January, 2020.

[36] E. Project, "Eclipse ide for java developers," https://www.eclipse.org/downloads/packages/release/2020-03/r, consulted July, 2020.

[37] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 209–219.

[38] L. Lavazza, D. Tosi, and S. Morasca, "An empirical study on the persistence of spotbugs issues in open-source software evolution," in *Quality of Information and Communications Technology*, M. Shepperd, F. Brito e Abreu, A. Rodrigues da Silva, and R. Pérez-Castillo, Eds. Cham: Springer International Publishing, 2020, pp. 144–151.

[39] L. N. Q. Do, J. R. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 835–847, 2020.

[40] H. Mohayeji, A. Agaronian, E. Constantinou, N. Zannone, and A. Serebrenik, "Securing dependencies: A comprehensive study of dependabot's impact on vulnerability mitigation," *Empirical Software Engineering*, vol. 30, no. 3, p. 89, 2025.

[41] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Understanding the evolution of android app vulnerabilities," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 212–230, 2019.

[42] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112.

[43] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats

in the npm ecosystem," in *28th USENIX Security symposium (USENIX security 19)*, 2019, pp. 995–1010.

[44] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in pypi ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 166–177.

[45] M. F. Rabbi, R. Paul, A. I. Champa, and M. F. Zibran, "Understanding software vulnerabilities in the maven ecosystem: Patterns, timelines, and risks," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 290–294.

[46] K. Jezek, J. Dietrich, and P. Brada, "How java apis break–an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[47] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice api evolution in practice: A study on strategies and challenges," *Journal of Systems and Software*, vol. 215, p. 112110, 2024.