computers

MDPI

Article

Design and Implement an Accurate Automated Static Analysis Checker to Detect Insecure Use of SecurityManager

Midya Alqaradaghi [1,2,*], Muhammad Zafar Iqbal Nazir [1] and Tamás Kozsik [1,*]

[1] Department of Programming Languages and Compilers, ELTE Eötvös Loránd University, Pázmány Péter stny. 1/C, H-1117 Budapest, Hungary; bsvncs@inf.elte.hu
[2] Technical Engineering College of Kirkuk, Northern Technical University, Kirkuk 36001, Iraq
* Correspondence: alqaradaghi.midya@inf.elte.hu (M.A.); kto@inf.elte.hu (T.K.)

Abstract: Static analysis is a software testing technique that analyzes the code without executing it. It is widely used to detect vulnerabilities, errors, and other issues during software development. Many tools are available for static analysis of Java code, including SpotBugs. Methods that perform a security check must be declared private or final; otherwise, they can be compromised when a malicious subclass overrides the methods and omits the checks. In Java, security checks can be performed using the SecurityManager class. This paper addresses the aforementioned problem by building a new automated checker that raises an issue when this rule is violated. The checker is built under the SpotBugs static analysis tool. We evaluated our approach on both custom test cases and real-world software, and the results revealed that the checker successfully detected related bugs in both with optimal metrics values.

Keywords: SecurityManager class; Java; SpotBugs; static analysis; accurate checker

Citation: 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. https://doi.org/10.3390/computers12120247

Academic Editor: Paolo Bellavista

Received: 6 November 2023
Revised: 19 November 2023
Accepted: 20 November 2023
Published: 28 November 2023

1. Introduction

Our existence is dependent on complex software systems that are challenging to ensure they are functioning properly. From a security point of view, a vulnerability is introduced when a coding or design error occurs [1]. Static analysis techniques detect security vulnerabilities early in software development, leading to time, effort, and cost savings [2–5]. Static analysis techniques can detect various security vulnerabilities and bugs, ranging from basic programming mistakes to more intricate issues such as access control problems [5]. Several static analysis tools exist for the most widely used programming languages, which can be integrated easily into the software development cycle and even made part of the continuous integration and continuous delivery CI/CD pipeline; this way, they can automate the discovery of many bugs and vulnerabilities.

An important class of security vulnerabilities in the Java programming scenario is related to the SecurityManager class (SM) in the standard application programming interface. This class provides means for applications to establish and enforce security policies. It enables an application to evaluate the nature of an operation, which may be unsafe or sensitive, before executing it. Doing so allows the application to determine if the operation can be carried out within the current security context. A security manager routine simply returns if the operation is permitted but throws a SecurityException if the operation is not permitted [6]. According to the SEI CERT Oracle Coding Standard for Java, methods that perform a security check must be declared private (which means that the method cannot be accessed from, or even overridden, from outside the class) or final (which means it cannot be overridden) [7] because otherwise, those member methods can be compromised when a malicious subclass overrides the methods and omits the checks. However, according to the same web page, no automated static analysis tool can detect non-final and non-private methods that perform security checks. Consequently, this paper targets this specific problem.

Many static analysis tools—with varying capabilities—can be employed for implementing our idea. We followed the following criteria to choose a suitable one. (1) It should be well-known, free, and open source, (2) target Java language, and (3) inspect the bytecode. We chose SpotBugs version 4.7.3 [8].

The motivations of this research are the following:

- Although losing a little popularity, Java is still among the most often-used programming languages in the software industry, according to the TIOBE index [9].
- Many applications used in our everyday lives are written in Java, with a long life span. Maintaining these applications and removing their weaknesses is essential.
- SM is common among the Java applications in GitHub repositories, but no practical work or research articles still focus on using it safely.
- Using static analysis techniques is effective in identifying security vulnerabilities and programming weaknesses.

The main contributions of the paper are the following:

- We build a novel checker named FindVulnerableSecurityCheckMethods under the SpotBugs static analysis tool that raises an issue when calling an overridable method that performs a security check;
- We evaluate our approach on custom test cases and real-world software, and report the evaluation results using four robust metrics: recall, false alarm rate (FAR), precision, and F1-score.

Problem Statement

According to the "SEI CERT Oracle Coding Standard for Java," it is advised to declare methods that perform security checks as either private or final [7]. This recommendation is made to safeguard these member methods from potential compromise. If these methods are not declared private or final, there is a risk that a malicious subclass could override them and deliberately omit the essential security checks, thereby introducing vulnerabilities.

Using SM to mitigate the potential of unintentional vulnerabilities in local code is practically difficult. There is much less usage of this class in production than most people realize, and many assertions that it is frequently used to secure local code have no evidence. Nowadays, SM is not used for some reasons: difficult programming model, poor performance, and weak permission model. Consequently, since the benefits of this class have significantly decreased over time due to changes in the deployment and threat environment, and due to the high cost of its maintenance, it has been deprecated for removal in a future release [10]. Still, it was necessary to conduct this research and build this checker because many Java applications perform security checks using the SM. We looked at Java software on GitHub and saw that 224 k files have been used [11]. Moreover, it is not always straightforward to upgrade applications to the newer Java version, as major version upgrades (e.g., upgrading from Java 8, or less, to Java 17) can require more effort and may involve addressing deprecated APIs, modifying code that relies on removed features, and adapting to changes in the language or libraries. This can involve code refactoring, updating dependencies, and addressing any compatibility issues or behavioral changes introduced in the newer version.

2. Related Work

According to the SEI CERT Oracle Coding Standard for Java, no automated static analysis tool can detect the call of overridable methods that perform security checks using one of the methods of the SM. However, some other checkers do similar work; they can detect the use of overridable methods in other insecure scenarios.

In this section, we present two categories of related works in a detailed and extensive way. The first are those static analyses that can detect similar issues to ours. The second category includes the research works that are interested in the security of the Java language, including the usage of the SM and the research papers that have used the class as a part of their implemented work.

In this first category, different commercial and free static analysis tools target the problem of calling overridable methods from other methods in Java code. Increasing the accessibility of overridden or hidden methods permits a malicious subclass to offer wider access to the restricted method than was originally intended [12]. This issue has been targeted by the Parasoft Jtest static analysis tool [13]. However, since the tool is commercial, the proposed solution is not in-handy, and we are unsure if it can detect our presented issue. SpotBugs, PVS-Studio [14], and SonarQube [15] all target the problem of calling an overridable method by a constructor. This problem may result in the use of uninitialized data, leading to runtime exceptions or to unanticipated outcomes. Calling overridable methods from constructors can also leak this reference before object construction is complete, which may expose the uninitialized or inconsistent data to other threads [16].

The clone method may invoke only methods that are non-overridable. The opposite is insecure because a malicious subclass could override the method and affect the behavior of the clone method. However, even if the subclass is trusted, it could observe (and possibly modify) the cloned object in a partially initialized state before its construction has concluded. In these cases, the subclass could leave the clone, the object being cloned, or both in an inconsistent state [17]. SpotBugs and Parasoft Jtest have targeted this problem.

None of the previously presented static analysis tools currently target the problem of calling overridable methods that use security checks. In our work, we targeted this issue and designed such a checker under SpotBugs. Table 1 presents the summary of the aforementioned static analysis checkers.

Table 1. Related work summary (first category).

The Issue	Tools Target It
Increasing the accessibility of overridden or hidden method	Parasoft Jtest
Calling an overridable method by a constructor	SpotBugs, SonarQube, PVS-Studio
Calling an overridable method by the method clone	SpotBugs, Parasoft Jtest
Calling an overridable method that performs a security check	Our checker, under SpotBugs

In the second category, we first give an overview of the research work that concerns the importance of Java security and/or the usage of the SM. Nevertheless, some of them shed light on the insecurity of Java, but not specifically the possible insecurity of this class.

Almut Herzog et al. [18] thoroughly analyzed the Java SM's performance, identifying permissions with subpar performance and making recommendations for improving efficiency. According to their experiments, resource access statement CPU execution penalties ranged from 5% to 100%. Some resource uses, such as file and socket access, can hide the performance cost of access control checks. Accessing main memory resources, however, results in observable latency penalties. They concluded that enhancing the policy file's garbage collection and authorization order is essential to achieving acceptable response times.

Joseph A. Bank [19] assessed the Java language's security concerns, its intended applications in Java-enabled web browsers, and Java's suggested fixes. He briefly described the history of executable contents, then talked about the potential security concerns associated with these contents and Java's responses to those risks, and then examined the efficacy of those solutions. He concluded that Java does increase the security risk, and the power gains must always be balanced against the security risk posed by Java-based systems. They also argued that using Java does not make sense for a system where security is of the utmost concern; the increased security risk is not worth it. Regarding home computers, they see that Java's advantages outweigh its drawbacks. Similarly, several systems are inaccessible online due to security concerns that exceed the advantages of using the internet.

In a study conducted in 1995 by A. Sterbenz [20], an introduction to the Java language was given, followed by a discussion of the problems that could occur when running unauthorized programs in an insecure setting. The efficiency and adaptability of the Java

security model were assessed in both theoretical and practical contexts across numerous application scenarios to find viable solutions to this issue. Their findings reveal several flaws in both the present implementations and the underlying security paradigm. The security features of the Java and. NET programming languages were compared by N. Paul et al. [21]. However, they discovered that, due to its more straightforward and cleanly designed architecture, NET is more secure than Java in several areas. In their analysis, they shed light on a variety of Java flaws, including the potential for a null SM. Rewriting the policy class or implementing a different SM may be required in Java to enhance the default policy implementation. Thereby, due to the enormous security concerns introduced by his level of extensibility, it is uncertain what advantages Java provides.

Secondly, we explain how numerous studies used the SM throughout the development stage of their software due to its different functions (for instance, login access, user management, message coding, and decoding). However, they did not consider the possibility of insecure use of its methods. Consequently, we briefly describe the related research to show the importance of implementing our checker.

G. Cabri et al. [22] used the Java authentication and authorization system (JAAS) to execute security checks on individual actions for each role that Java agents play and concluded that it outperformed the usage of the SM. P.J. Clemente et al. [23] have provided a new method for creating fully Model-Driven Architecture (MDA) compatible component-and-aspect-based software systems. With this method, the system can be modeled at several levels of abstraction, from early platform agnostic models through platform-specific models and towards the system's final code. The SM component has addressed the security issue during its development process, which has also been incorporated into the system to grant authorized access to the online store.

In order to assess the benefits of Aspect-Oriented Software Development (AOSD) for Model-Driven Architecture (MDA), B. Tekinerdogan et al. [24] have provided a thorough analysis of cross-cutting issues in the context of the MDA. Based on their research, they provided a few recommendations regarding Java and the process used in the MDA approach. According to a chosen list of cross-cutting problems, they have identified several evolutionary possibilities. All the designed scenarios have used SM, although some of these scenarios have upgraded it.

Also, SM was employed in JoiN (a Java-based software framework for building massively parallel grids that can run huge parallel applications) by E.J.H. Yero et al. [25] to offer an access control mechanism based on roles that are integrated throughout the internal structure of the system. Furthermore, JoiN has advanced fault tolerance characteristics that enable it to withstand failures in both the computers handling the groups and the computers executing parallel tasks. JoiN employs a dynamic, adaptable scheduling method that repeats parallel activities for fault tolerance and variations in resource availability.

M.C. Little [26] created W3OTrans, a tool that enables programmers to build web apps utilizing atomic actions and objects. The C++ and Java implementations of W30Trans allow server-side programs to utilize more effective compilation methods fully. However, using Java poses some significant security concerns. They used the SM object to impose security, which controls what an applet can and cannot do.

Our work differs from the previously presented works in that we considered the importance of using the methods of the SM in an insecure way; consequently, we designed and implemented a free and open-source static analysis checker that targets this issue. Table 2 presents the summary of these research papers, focusing on the SM related information.

Table 2. Related work summary (second category).

References	Concern About		Use the SM in Implementing Their Ideas	Research Conclusions
	Java Security	SM		
[18,21]	Yes	Yes - Analyzed the SM's performance - Focused on Java flaws, including the potential for a null SM	No	- The policy file's garbage collection should be improved - NET is more secure than Java in several areas - Implementing a different SM may be required in Java to enhance the default policy implementation
[19,20]	Yes	No	No	- Java does increase the security risk - There are several flaws in the underlying security paradigm of Java
[22–26]	No	No	Yes - To address the security issues during the development process - To offer an access control mechanism - To impose security - Used to design scenarios	- Using the JAAS to achieve security checks outperformed the usage of the SM
Current	No	Yes - Design and implement a checker that detects an overridable method's calling that performs a security check	Yes - To build the custom test cases	- The designed checker gave the optimal performance

3. Research Methodology

In this section, we explain our method of designing the checker, followed by the method of building the custom test cases, and finally, we explain the checker's performance evaluation method and the used metrics.

3.1. Checker Design

Figure 1 presents the detailed method process we followed for executing our checker named (FindVulnerableSecurityCheckMethods). Since the methods of final classes cannot be overridden, the usage of SM could be insecure only if used in a non-final class; consequently, we designed our checker to focus only on those classes. For each non-final class, it inspects for method invocations, followed by searching for neither final nor private methods because methods that are either final or private cannot be overridden. Subsequently, an issue will be reported if the invoked method performs a security check using one of the methods of the SM. The previously explained method has been implemented successfully under the SpotBugs static analysis tool and passed the internal review of our team; see reference [27] for more details concerning the implementation coding. Our approach was accepted by the official SpotBugs software as a novel checker [28] and has been included in SpotBugs version 4.8.0.

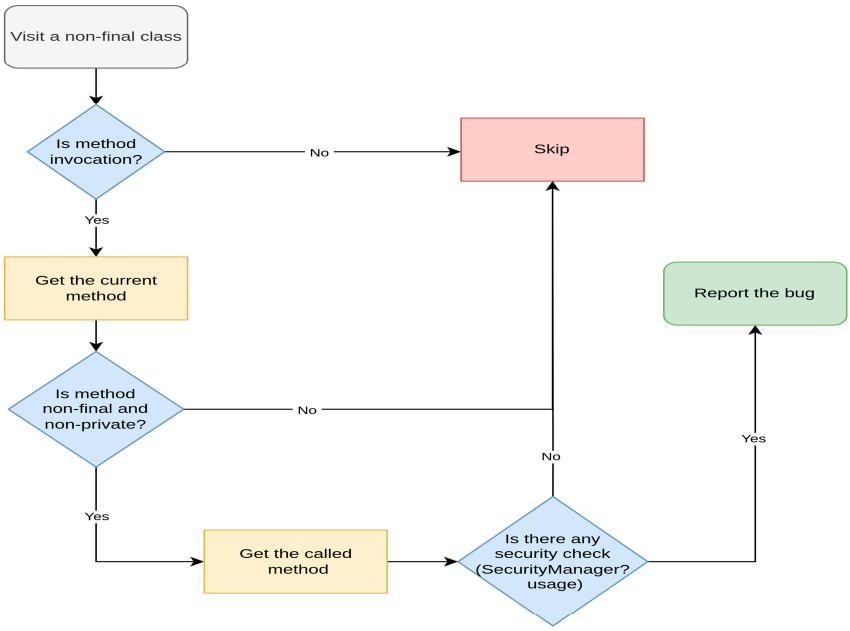

Figure 1. The execution of our checker.

3.2. Custom Test Cases Design

To evaluate our produced checker and comprehensively cover the issue under the study, our team designed custom test cases that cover the following criteria:

- Non-compliant (NC) and compliant (C): we designed flawed and non-flawed test cases to be able to test our checker performance in both true positive (TP) and false positive (FP) aspects;
- Unambiguous: they are written clearly and concisely, leaving no room for misinterpretation;
- Test objective: each test case has a specific objective clearly stating what aspect of the issue is being tested;
- Independence: For better isolation and identification of issues, we designed independent test cases;
- Validated expectations: each test case has an expected outcome defined beforehand.

Consequently, we came out with 6 NC and 18 C test cases. Next, we give details about their design.

NC test cases: these test cases should be reported by our checker, as they include real bugs and are considered insecure. By designing these six test cases, we could cover all the possible scenarios when SM methods are insecure, i.e., a SM method is used in a non-final and non-private method and included in a non-final class. Table 3 presents the details of these test cases. Code snippet 1 presents test case 4 (T4).

Table 3. NC test cases description.

Test Cases	Description
T1	Using a named SM object
T2	Using a nameless SM
T3	Using a protected method
T4	Calling a vulnerable method from another method.
T5	Using obvious true if statement (ex: if (5 > 3))
T6	Using arbitrary if statement (if (ex: x < 12), when the value of x is unknown)

Code snippet 1. NC test case. Calling a non-final and non-private method uses an SM object to perform a security check from another method.

```
public class FindVulnerableSecurityCheckMethodsTest {
    Logger logger = Logger.getAnonymousLogger();

    public void badCallee() {
        (new FindVulnerableSecurityCheckMethodsTest()).badCalled();
    }
    public void badCalled() {
        try {
            SecurityManager sm = System.getSecurityManager();
            if (sm! = null) {
                sm.checkRead("/temp/tempFile");
            }
            //Access the file
        } catch (SecurityException se) {
            logger.log(Level.WARNING, se.toString());
        }
    }
}
```

C test cases: our checker should not report these test cases, as they do not include real bugs; rather, they include safe coding scenarios to test if the static analysis (our checker) successfully avoids them. We designed two test case categories. In the first one, non-final classes were used, and are presented in Table 4; however, in the second category, we used a combination of test cases from the first category and the NC test cases, but we put them in final classes, which is enough to make them secure. By designing these eighteen test cases, we could cover all the possible scenarios when SM methods are secure. Code snippets 2 and 3 are examples of test cases of both categories (T1 and T4).

Table 4. C test case description (first category).

Test Cases	Description
T1	Using a final method
T2	Using a private method
T3	Creating a SM object without using it
T4	Performing the security check in an obvious false branch (like if (5 > 10))
T5	Calling a method of SM that does not perform any security checks
T6	Using a user-defined SM instead of java.lang.SecurityManager

Code snippet 2. C test case (1st category). A final class includes a non-final and non-private method that uses a SM object to perform a security check.

```
public final class GoodSecurityCheckMethodsTest1{
    Logger logger = Logger.getAnonymousLogger();

    public void goodSecurityCheckMethodsTestCheck(){
        try {
            SecurityManager sm = System.getSecurityManager();
            if (sm! = null) {//Check for permission to read file
                sm.checkRead("/temp/tempFile");
            }
            //Access the file
        } catch (SecurityException se) {
            logger.log(Level.WARNING, se.toString());
        }
    }
}
```

Code snippet 3. C test case (2nd category). A non-final class includes a public method that uses a SM object, but it is not vulnerable because of performing the security check in an obvious false condition branch.

```
public class GoodSecurityCheckMethodsTest2{
    Logger logger = Logger.getAnonymousLogger();

    public void goodSecurityCheckMethodsTestCheck6(){
        try {
            SecurityManager sm = System.getSecurityManager();

            if (sm! = null) {//Check for permission to read file
                if (5> 10) {
                    sm.checkRead("/temp/tempFile");
                }
            }
            //Access the file
        } catch (SecurityException se) {
            logger.log(Level.WARNING, se.toString());
        }
    }
}
```

You can refer to the second author's GitHub repository [27] for checking the other NC and C test cases. NC test cases contain the word bad in the test method or the file name, while C test cases contain the word good.

3.3. Performance Evaluation

For the evaluation purpose, we used four metrics, namely, recall, false alarm rate (FAR), precision, and F1-score, to represent distinct aspects of our checker's performance in identifying the target issue of this paper. We first ran our checker on the test cases, then computed the number of TP, and FP to determine these metrics:

$$\text{Recall} = \frac{\text{TP}}{\text{NC testcases}} \tag{1}$$

$$\text{FAR} = \frac{\text{FP}}{\text{C testcases}} \tag{2}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3}$$

$$\text{F1} - \text{score} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{4}$$

True positives (TP) are NC constructs correctly classified as NC. False positives (FP) are C constructs misclassified as NC ones. Recall is expressed as the quotient of TP to the total number of NC test cases (Equation (1)). Hence, recall is restricted to NC constructs and represents the percentage of NC test cases accurately identified by our checker. FAR provided in Equation (2) is exclusively concerned with C test cases, and it reflects the percentage of C test cases that have been misidentified as NC ones. It is the ratio of FP to the total number of compliant test cases or the ratio of mistakenly reporting C constructions as NC ones.

Precision, as defined in Equation (3), calculates the ratio of TP to the sum of TP and FP. Therefore, precision encompasses all reported test cases and represents the percentage of accurately identified NC constructs out of the total number of constructs reported by a tool.

As defined in Equation (4), the F1-score combines recall and precision by taking their harmonic mean. This allows us to merge these two important metrics into a single measure.

In this paper, we used it to describe the accuracy of the analysis. We did not use the standard accuracy metric, which is the fraction of correct detections (TP + TN (TN: true negatives are compliant constructs correctly ignored by a static analysis tool.)) out of all detections (TP + TN + FP + FN (FN: false negatives are compliant constructs mistakenly detected by static analysis.)). This metric is suitable for balanced classification, but in our experiment, the numbers of NC and C occurrences are imbalanced, both for the custom test cases and the real-world software.

High values of recall, precision, the F1-score, and low values of the FAR indicate the best performance. All falling in the interval {0, 1}.

4. Results

The result of running our checker on related Java bugs and the performance evaluation results of the checker are presented in this section. Two Java bug types were used here; the first type is those built with intentional flaws (custom test cases), and the second type is real-world bugs.

4.1. Analyzing Custom Test Cases

In this section, the results of running our checker on the custom test cases, which were explained in Section 3.2, are presented in Table 5. Our checker could find all the NC test cases and correctly ignore all the C test cases, resulting in TP = 6 and FP = 0. Moreover, the recall, FAR, precision, and F1-score values were optimal (i.e., 1.0, 0.0, 1.0, and 1.0), respectively.

Table 5. Evaluation results of analyzing the custom test cases.

Metrics	Value
NC test cases	6
C test cases	18
TP	6
FP	0
Recall	1.00
FAR	0.00
Precision	1.00
F1-score	1.00

4.2. Analyzing Real Software

To evaluate our checker performance in detecting the related bug in real-world software, we analyzed nine pieces of software: SpotBugs itself, Elasticsearch [29], selective-security-manager [30], Lottie Android [31], mybatis-3 [32], tomcat [33], intellij-community [34], tutorials [35], and netty [36]. These software programs were not selected arbitrarily; we selected those that use SM.

We could calculate the number of C and NC occurrences in this software using a simple grep command; consequently, the metrics values were calculated for evaluating the checker performance on detecting the related flaws in the analyzed software. The evaluation results are presented in Table 6; they revealed that our checker gave optimal performance values.

There were no NC cases for netty software, and the checker successfully did not detect any flaws (TP and FP = 0), resulting in recall and precision = N/A, because both equations' denominators are zeros. Consequently, the value of F1-score was N/A, for the same reason.

Table 6. Evaluation results of analyzing the real-world software.

Metrics	SpotBugs	Elasticsearch	Selective-Security-Manager	Lottie Android	Mybatis-3	Tomcat	Intellij-Community	Tutorials	Netty
NC	6	8	4	1	1	7	6	2	0
C	0	7	0	0	0	1	4	1	2
TP	6	8	4	1	1	7	6	2	0
FP	0	0	0	0	0	0	0	0	0
Recall	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	N/A
FAR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00
Precision	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	N/A
F1-score	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	N/A

5. Conclusions

In the presented study, we designed and implemented a novel checker named FindVulnerableSecurityCheckMethods under the SpotBugs static analysis tool, which can detect the calling of overridable methods that perform security checks in Java programs.

We evaluated our approach on both custom test cases and real-world software. The results revealed that the produced checker is accurate and precise when detecting the related issue in the analyzed programs. The performance metrics values were optimal for all the analyzed test cases and real-world software.

6. Future Work

In order to properly validate our approach, we will examine more real-world software in future studies. The checker's design may be improved by including a feature that offers substitutes for carrying out SecurityManager's work. This is in line with the class being removed in the subsequent Java version.

Author Contributions: M.A.: writing the manuscript, methodology, checker design, custom test cases, and performance evaluation. M.Z.I.N.: checker implementation, custom test cases, running analysis, contact for SpotBugs public review, and related technical issues. T.K.: review and editing, and supervision. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by project no. TKP2021-NVA-29 under the Ministry of Innovation and Technology of Hungary from the National Research, Development, and Innovation Fund and financed under the TKP2021-NVA funding scheme.

Data Availability Statement: The data that support the findings of this study are available from the corresponding author M.A., upon request.

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

References

1. Algarni, A.M. The Historical Relationship between the Software Vulnerability Life cycle and Vulnerability Markets: Security and Economic Risks. *Computers* **2022**, *11*, 137. [CrossRef]
2. Amankwah, R.; Chen, J.; Song, H.; Kudjo, P.K. Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites. *Softw.-Pract. Exp.* **2023**, *53*, 1125–1143. [CrossRef]
3. Sun, P.; Kim, D.K.; Ming, H.; Lu, L. Measuring Impact of Dependency Injection on Software Maintainability. *Computers* **2022**, *11*, 141. [CrossRef]

4. Alqaradaghi, M.; Kozsik, T. Inferring the Best Static Analysis Tool for Null Pointer Dereference in Java Source Code. *CEUR Workshop Proc.* **2022**, *3237*, 1–12.
5. Chess, B.; West, J. *Secure Programming with Static Analysis*; Addison-Wesley: Reading, MA, USA, 2007; p. 11.
6. Java Documentation, Class SecurityManager. Available online: https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html (accessed on 30 September 2023).
7. SEI CERT Oracle Coding Standard for Java, MET03-J: Methods that Perform a Security Check Must Be Declared Private or Final. Available online: https://wiki.sei.cmu.edu/confluence/display/java/MET03-J.+Methods+that+perform+a+security+check+must+be+declared+private+or+final (accessed on 30 September 2023).
8. SpotBugs, Find Bugs in Java Programs. Available online: https://spotbugs.github.io/ (accessed on 20 September 2023).
9. TIOBE Index for October 2023. Available online: https://www.tiobe.com/tiobe-index (accessed on 2 November 2023).
10. Java Documentation, JEP 411: Deprecate the Security Manager for Removal. Available online: https://openjdk.org/jeps/411 (accessed on 30 September 2023).
11. Searching for SecurityManager in GitHub. Available online: https://github.com/search?q=SecurityManager+language%253AJava&type=code (accessed on 2 November 2023).
12. SEI CERT Oracle Coding Standard for Java, MET04-J. Do Not Increase the Accessibility of Overriden or Hidden Methods. Available online: https://wiki.sei.cmu.edu/confluence/display/java/MET04-J.+Do+not+increase+the+accessibility+of+overridden+or+hidden+methods (accessed on 1 November 2023).
13. Parasoft Jtest, Automated Java Software Testing and Static Analysis. Available online: https://www.parasoft.com/ (accessed on 1 November 2023).
14. PVS-Studio, Static Analyzer on Guard of Code Quality, Security (SAST), and Code Safety. Available online: https://pvs-studio.com/en/ (accessed on 1 November 2023).
15. SonarQube, A Self-Managed, Automatic Code Review Tool that Systematically Helps You Deliver Clean Code. Available online: https://docs.sonarsource.com/sonarqube/latest/ (accessed on 1 November 2023).
16. SEI CERT Oracle Coding Standard for Java, MET05-J. Ensure that Constructors Do Not Call Overridable Methods. Available online: https://wiki.sei.cmu.edu/confluence/display/java/MET05-J.+Ensure+that+constructors+do+not+call+overridable+methods (accessed on 1 November 2023).
17. SEI CERT Oracle Coding Standard for Java, MET05-J. Don Not Invoke Overridable Methods in Clone(). Available online: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487921 (accessed on 1 November 2023).
18. Herzog, A.; Shahmehri, N. Performance of the Java security manager. *Comput. Secur.* **2005**, *24*, 192–207. [CrossRef]
19. Joseph, A.B. Java Security, Computer Based Learning Unit. University of Leeds, Leeds, UK, 8 December 1995. Available online: https://groups.csail.mit.edu/mac/users/jbank/javapaper/javapaper.html (accessed on 1 November 2023).
20. Sterbenz, A. An evaluation of the java security model. In Proceedings of the IEEE Proceedings 12th Annual Computer Security Applications Conference, San Diego, CA, USA, 9–13 December 1996.
21. Paul, N.; Evans, D. Comparing Java and .NET security: Lessons learned and missed. *Comput. Secur.* **2006**, *25*, 338–350. [CrossRef]
22. Cabri, G.; Ferrari, L.; Leonardi, L. Applying security policies through agent roles: A JAAS based approach. *Sci. Comput. Program.* **2006**, *59*, 127–146. [CrossRef]
23. Clemente, P.J.; Hernández, J.; Conejero, J.M.; Ortiz, G. Managing crosscutting concerns in component based systems using a model driven development approach. *J. Syst. Softw.* **2011**, *84*, 1032–1053. [CrossRef]
24. Tekinerdogan, B.; Aksit, M.; Henninger, F. Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach. *Electron. Notes Theor. Comput. Sci.* **2007**, *163*, 45–64. [CrossRef]
25. Yero, F.I.H.; De Oliveira Lucchese, F.; Sambatti, F.S.; Von Zuben, M.; Henriques, M.A.A. JoiN: The implementation of a Java-based massively parallel grid. *Futur. Gener. Comput. Syst.* **2005**, *21*, 791–810. [CrossRef]
26. Little, M.C. Constructing reliable Web applications using atomic actions. *Comput. Networks* **1997**, *29*, 1281–1290. [CrossRef]
27. MET03-J Checker. Available online: https://github.com/NazirMuhammadZafarIqbal/spotbugs/tree/MET03 (accessed on 15 October 2023).
28. Public Review of MET03-J Checker. Available online: https://github.com/spotbugs/spotbugs/pull/2447 (accessed on 25 September 2023).
29. Elasticsearch: Free and Open, and Distributed, RESTful Search and Analytics Engine at the Heart of the Elastic Stack. Available online: https://github.com/elastic/elasticsearch (accessed on 20 October 2023).
30. Alphaloop Selective Security Manager: Demonstrates How to Create a Java SecurityManager that Can Be Enabled and Disabled Programmatically on Specific Threads. Available online: https://github.com/alphaloop/selective-security-manager (accessed on 20 October 2023).
31. Lottie Android, Render After Effects Animations Natively on Android and iOS, Web, and React Native. Available online: https://github.com/airbnb/lottie-android (accessed on 20 October 2023).
32. Mybatis—3, MyBatis SQL Mapper Framework for Java. Available online: https://github.com/mybatis/mybatis-3 (accessed on 15 October 2023).
33. Tomcat, Apache Tomcat Fork that Allows Tomcat Web Applications to Be Written and Run as Jigsaw Modules. Available online: https://github.com/pjBooms/tomcat (accessed on 15 October 2023).

34. Intellij—Community, IntelliJ IDEA Community Edition. Available online: https://github.com/Randgalt/intellij-community (accessed on 15 October 2023).
35. Tutorials, Just Announced—"Learn Spring Security Oauth". Available online: https://github.com/virtuoushub/tutorials (accessed on 15 October 2023).
36. Netty, Netty Project—An Event-Driven Asynchronous Network Application Framework. Available online: https://github.com/derklaro/netty (accessed on 10 October 2023).